INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad -500 043

COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES

by

B Padmaja

Associate Professor, CSE

Module – 1
Introduction to Data Structures, Searching and Sorting

Contents

- Introduction to Data Structures
- Classification of Data Structures
- Operations on Data Structures
- Searching Techniques - Linear, Binary
- Sorting Techniques- Bubble, Selection, Insertion,
- Comparison of Sorting Algorithms

Introduction to Data Structures

Data structure is a method of **organizing** large amount of data more **efficiently** so that any operation on that data becomes **easy**.

To develop a **program** of an algorithm we should select an appropriate **data structure** for that algorithm.

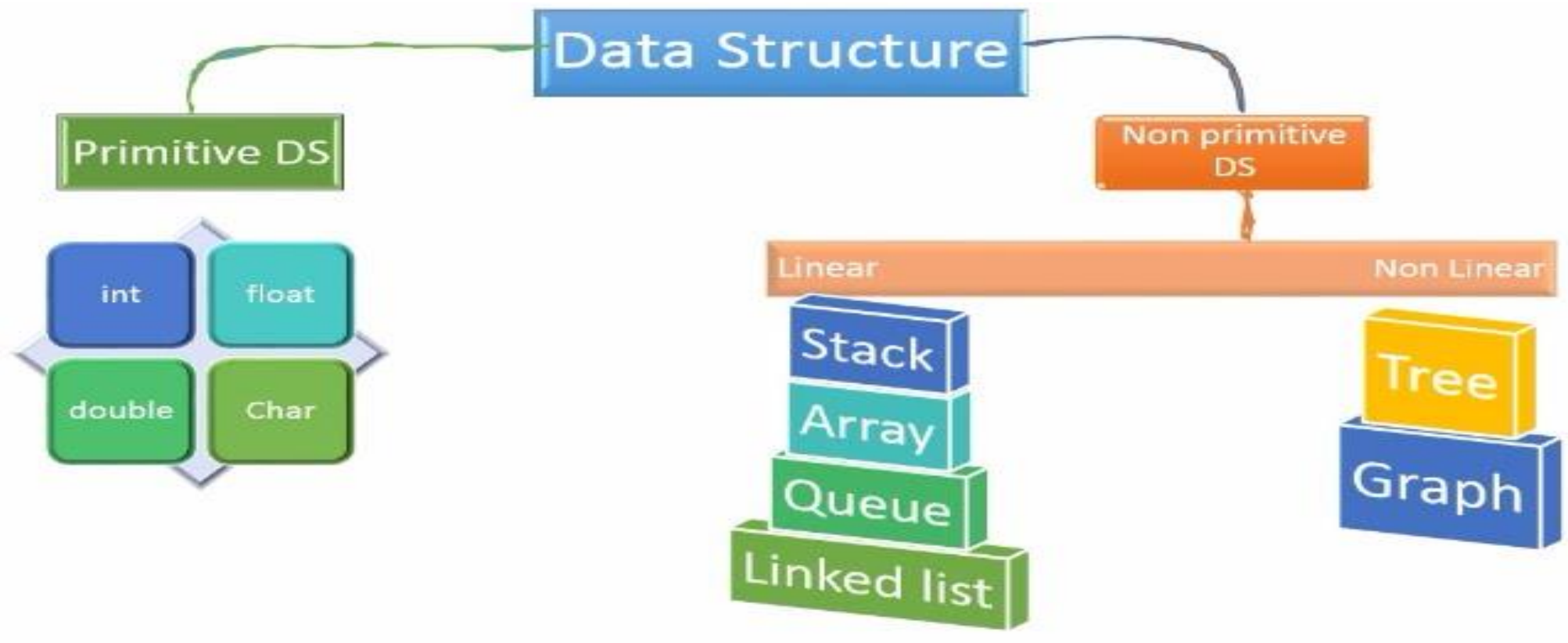
Algorithm + Data Structure = Program

Every data structure is used to organize the large amount of data.

Every data structure follows a particular principle.

The operations in a data structure should not violate the basic principle of that data structure.

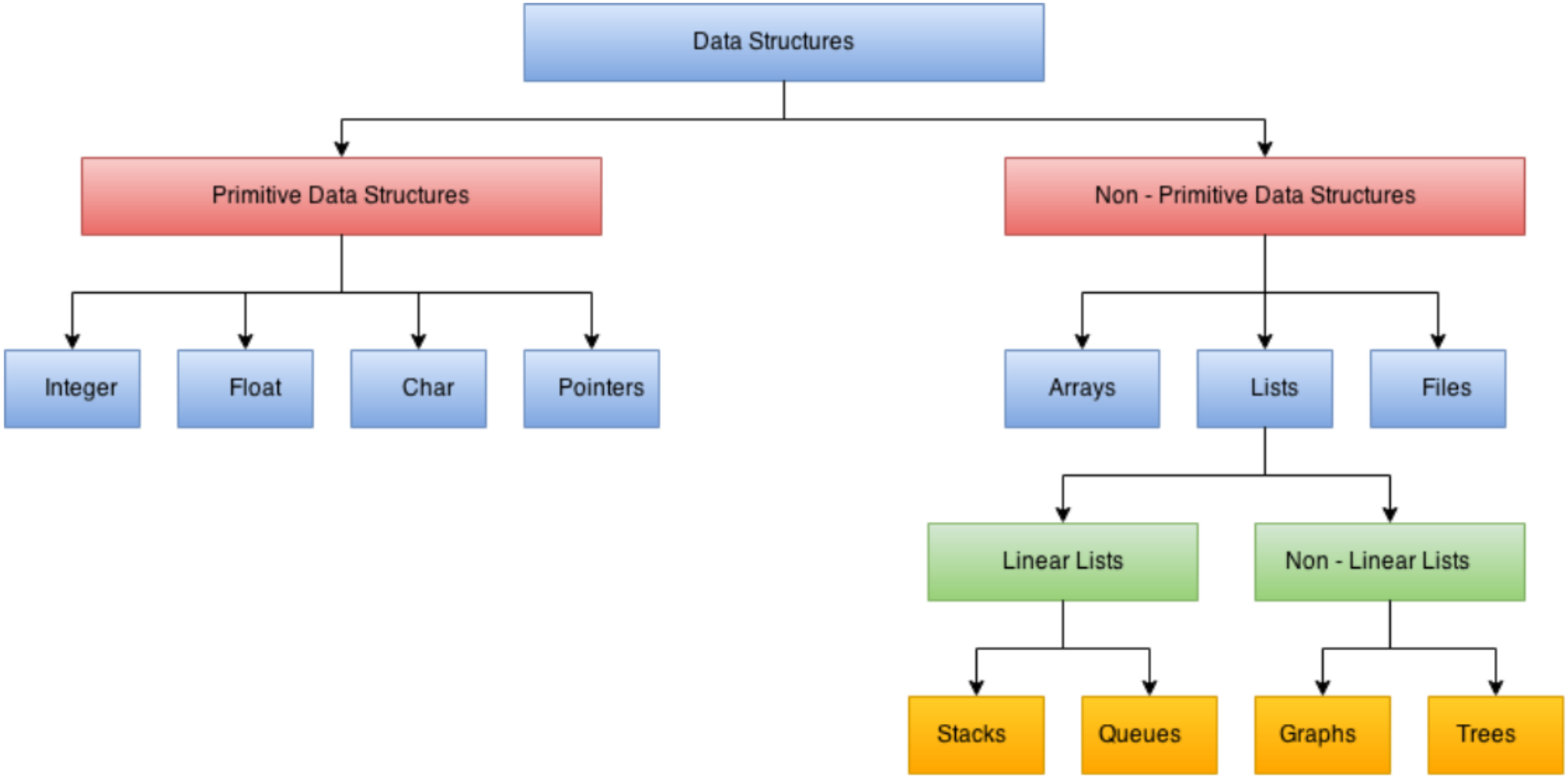
Classification of Data Structures



Classification of Data Structures

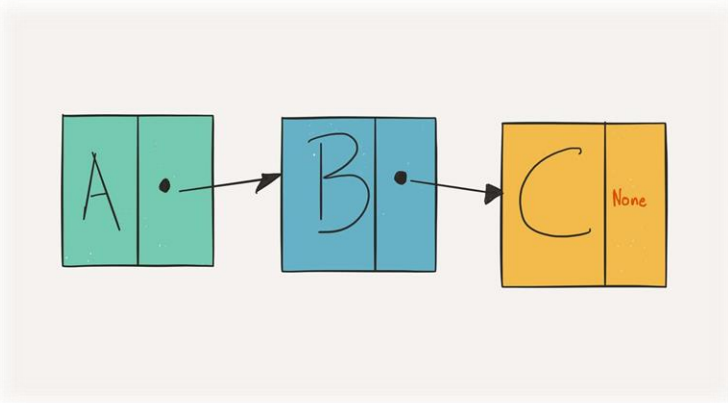
Primitive data structures

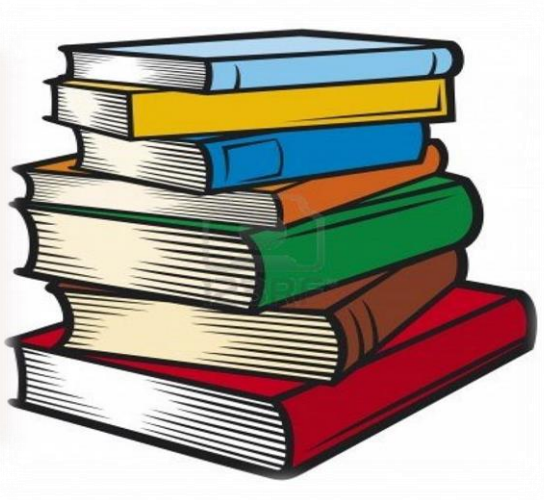
Non-primitive data structures



Linear Data Structures

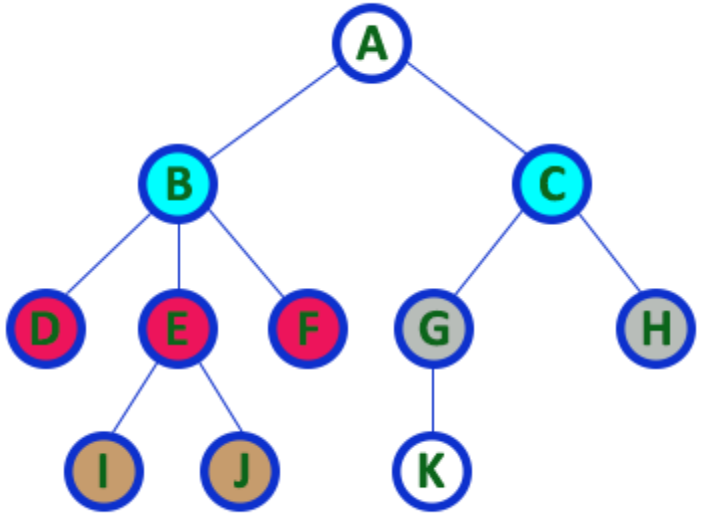
If a Data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure.

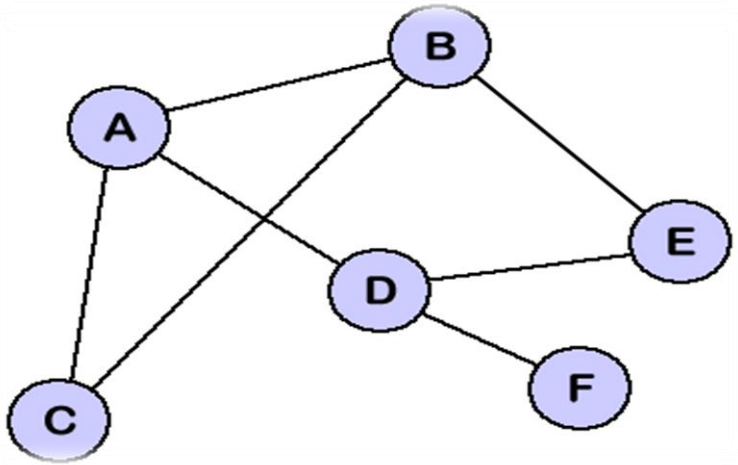




Non-linear Data Structures

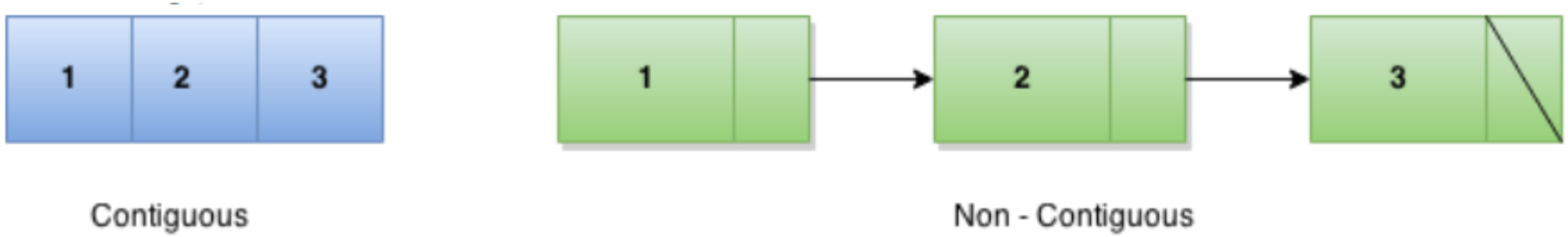
If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure.





Classification of Data Structures – Organization of data

- Contiguous Data Structures
- Non-contiguous Data Structures

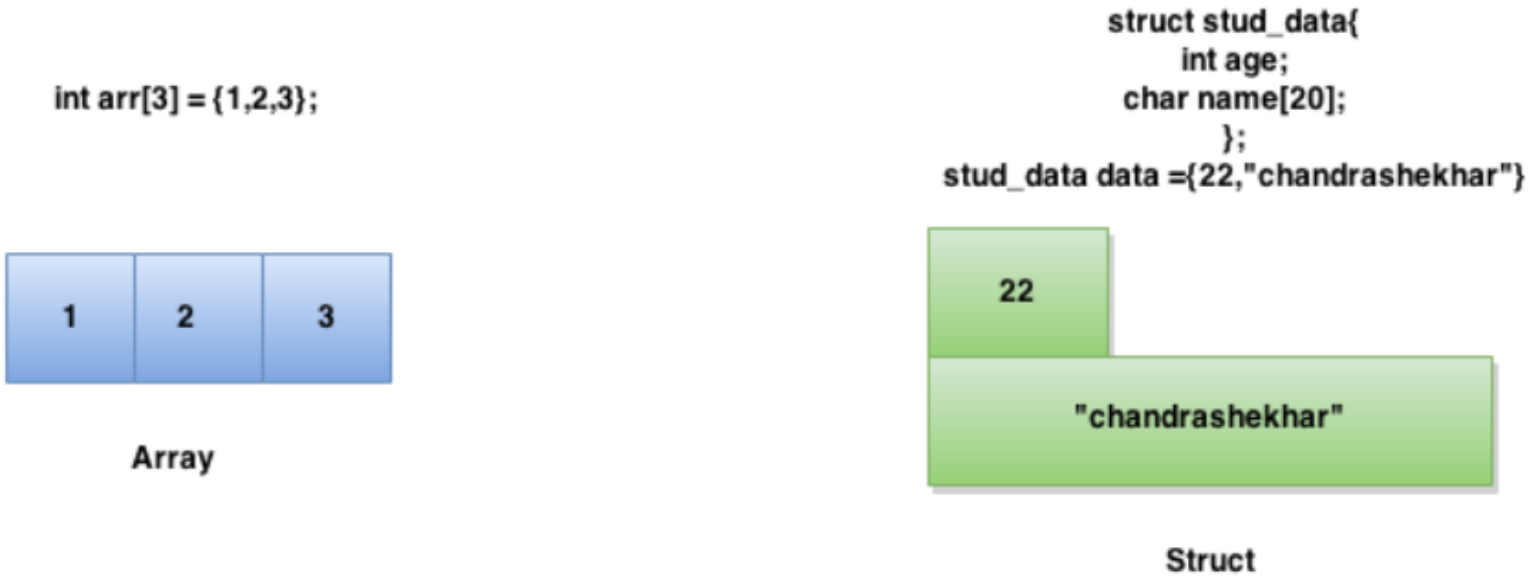


- Contiguous Data Structures - Ex. Arrays
- Non-contiguous Data Structures – Ex. Linked Lists

Contiguous Data Structures

Contiguous Data Structures are of 2 types:

- Arrays: Which contains data items of all the same size.
- Structures: Which contains data items of different size.



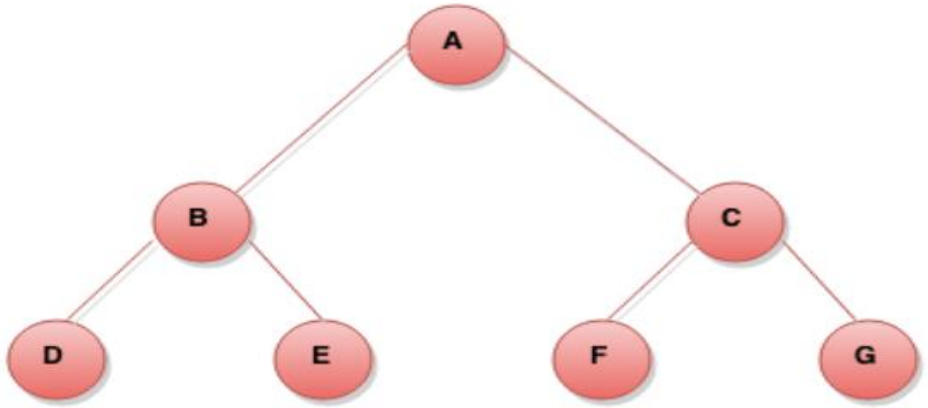
contiguous structures example

Non- Contiguous Data Structures

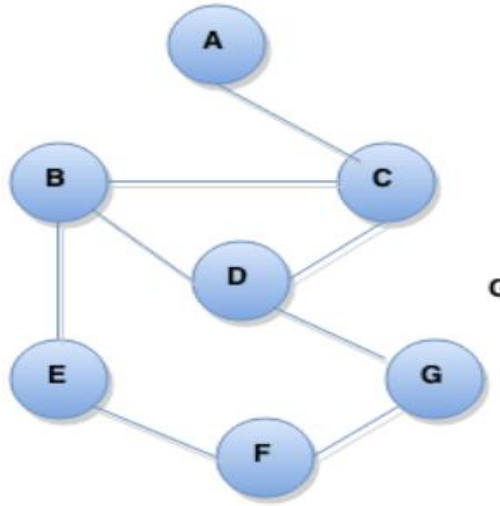
Non-Contiguous Data Structures:

- Linked-lists: It is a linear, one-dimensional type of non-contiguous DS.
- Trees and Graphs: It is a two-dimensional non-contiguous DS.

Linked List



Tree



Graph

Operations on Data Structures

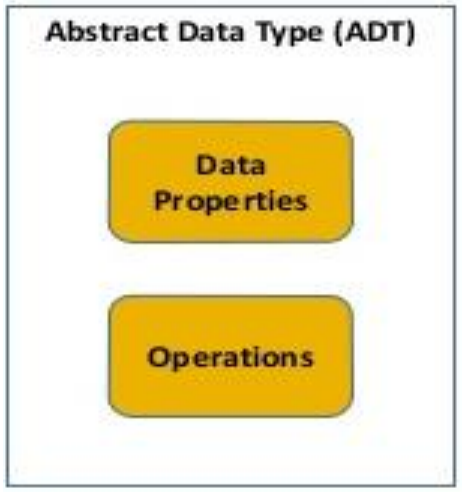
Traversing:	• Accessing each record exactly once so that certain items in the record may be processed.
Searching:	• Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.
Inserting:	• Adding a new record to the structure.

Operations on Data Structures

Deleting:	• Removing a record from the structure.
Sorting:	• Arranging the records in some logical order.
Merging:	• Combining the records in two different sorted files into a single sorted file.

Abstract Data Type (ADT)

ADT is a mathematical model with collection of operations defined on that model.



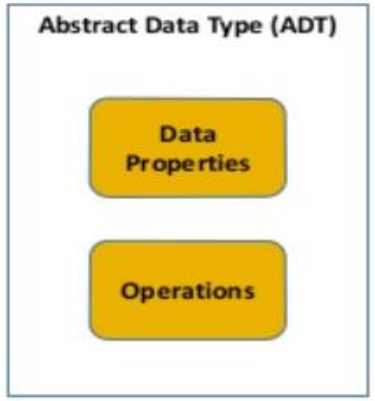
ADT separates data type declaration from representation.

ADT separates function declaration (prototype) from implementation.

Abstract Data Type (ADT)

Abstract Data Type

- **Model** of a data type
  - Properties of the data
  - Operations that can be performed on that data
- ***Definition:*** *Abstract data type* (ADT) is a **mathematical model** with a collection of operations defined on that model.

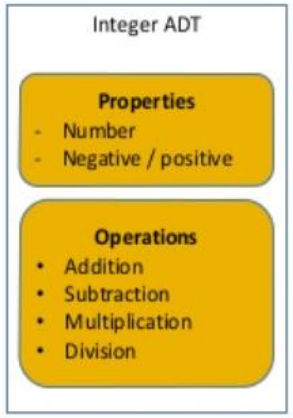


Implementation of ADT:

Implement the operations of a Data Structure using a programming Language.

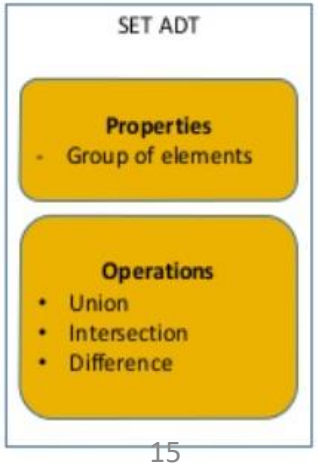
Example of Abstract Data Type (ADT)

- Integer
  - ...., -4, -3, -2, -1, 0, 1, 2, 3, 4 ...



One more Example of ADT

- SET
  - {2,4,6,8,10}
- Take two SET as input and return union
  - SET union(SET a, SET b)
- Take two SET as input and return intersection
  - SET intersection(SET a, SET b)
- Take two SET as input and return difference
  - SET difference(SET a, SET b)



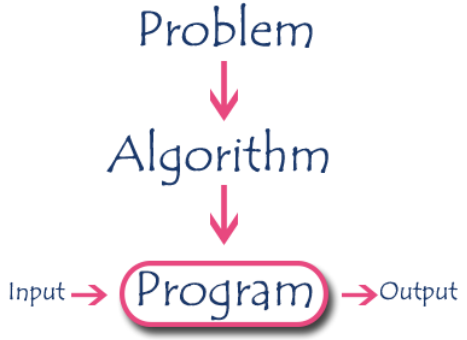
Abstract Data Type (ADT) Implementation

There are two basic structures we can use to implement an ADT List: arrays and linked lists.

- Array Implémentation
- Linked List Implémentation

Algorithm and its properties

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer



Algorithm Specifications

Input	Output	Definiteness	Finiteness	Effectiveness
• Every algorithm must take zero or more number of input values from external.	• Every algorithm must produce an output as result.	• Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation)	• For all different cases, the algorithm must produce result within a finite number of steps.	• Every instruction must be basic enough to be carried out and it also must be feasible.

Example of an Algorithm

Problem Statement : Find the largest number in the given list of numbers?

Input : A list of positive integer numbers.

Output : The largest number in the given list of positive integer numbers.

Example of an Algorithm

Algorithm	Code in C Programming

Step 1: Define a variable '**max**' and initialize with '**0**'.
 	
Step 2: Compare first number (say '**x**') in the list '**L**' with '**max**', if '**x**' is larger than '**max**', set '**max**' to '**x**'.
 	
Step 3: Repeat step 2 for all numbers in the list '**L**'.
 	
Step 4: Display the value of '**max**' as a result.
	int findMax (L)
	{
	int max = 0,i;
	for(i=0; i < listSize; i++)
	{
	if(L[i] > max)
	max = L[i];
	}
	return max;
	}

Approaches to Design an Algorithm

General approaches to the construction of efficient solutions to problems.

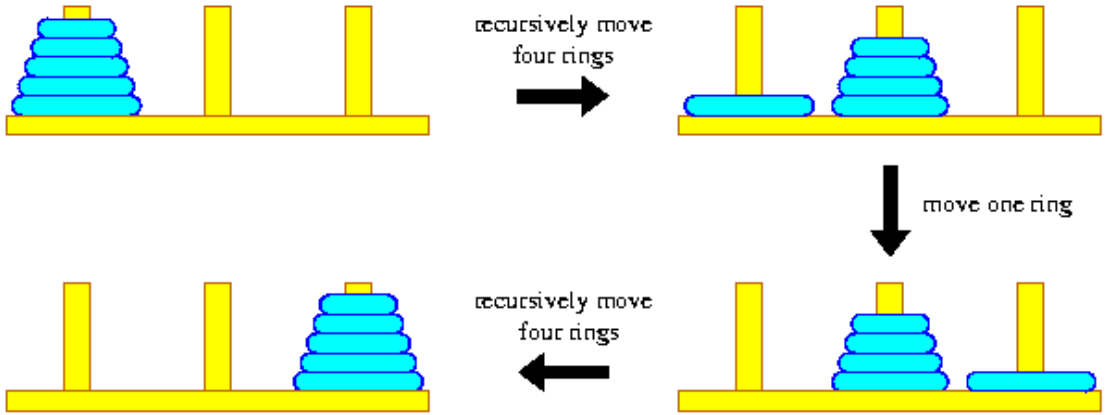
Brute force	• a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved.
Greedy	• The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. At each step the choice must be locally optimal – this is the central point of this technique.

Approaches to Design an Algorithm

Divide-and-Conquer	• Given an instance of the problem to be solved, split this into several smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.
Dynamic Programming	• The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice. The method usually accomplishes this by maintaining a table of sub-instance results.
Backtracking	• we start with a possible solution, which satisfies all the required conditions. Then we move to the next level and if that level does not produce a satisfactory solution, we return one level back and start with a new option.
Branch and Bound	• The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth–first search.

Recursive Algorithms

The function which calls by itself is called as Direct Recursive function (or Recursive function)



- An algorithm is called **recursive** if it solves a problem by reducing it to an instance of the same problem with smaller input.

Example: Factorial of a number

Computing n!

Procedure factorial (n : nonnegative integer)

if n= 0 then factorial(n)=1

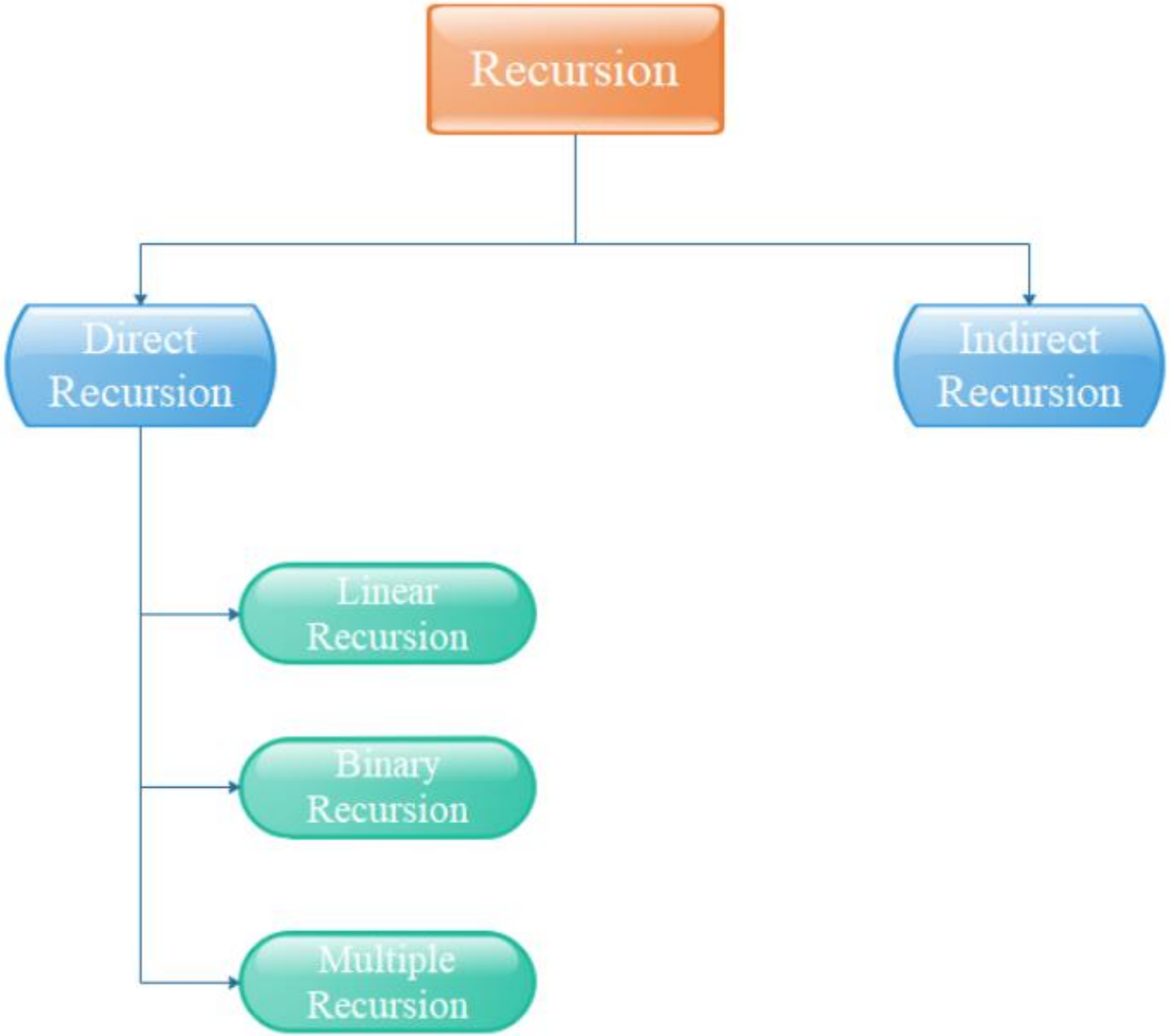
else factorial(n) := n * factorial(n-1)

Take n=4

4! = 4*3! = 4 * (3 *2!) = 4*3*2*(1!) = 4*3*2*1*(0!) = 24

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

Types of Recursion



Types of Recursion

- Linear Recursion: makes at most one recursive call each time it is invoked. E.g. factorial, sum of natural numbers, GCD
- Binary Recursion: makes two recursive calls. E.g. Fibonacci series
- Multiple Recursion: makes more than two recursive calls. E.g. Combinatorial puzzles
- Tail Recursion: A recursive function call is tail recursive when recursive call is the last thing executed by the function. E.g. factorial of a number.

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise.} \end{cases}$$

pot + pan = bib

dog + cat = pig

boy + girl = baby

Recap

1. Which of the following data structure is non-linear type?
   a. Strings
   b. Stacks
   c. Linked lists
   d. Trees

2. Which of the following data structure is linear type?
   a. Strings
   b. Queues
   c. Linked lists
   d. All the above

Recap

3. To represent hierarchical relationship between elements, which data structure is suitable?

a. Queues
b. Stacks
c. Trees
d. All the above

4. An algorithm that calls itself directly or indirectly is known as?

a. Sub algorithm
b. Recursion
c. Traversal algorithm
d. All the above

Recap

5. Which of the following are algorithmic design techniques?
   a. Divide-and-conquer
   b. Branch and bound
   c. Dynamic programming
   d. All the above

6. Which is not a property of an algorithm?
   a. Generality
   b. Effectiveness
   c. Definiteness
   d. Performance

Recap

7. LIFO (Last in first out) mechanism is used in which data structure?
   a. Linked lists
   b. Arrays
   c. Queues
   d. Stacks

8. FIFO (First in first out) mechanism is used in which data structure?
   a. Linked lists
   b. Arrays
   c. Queues
   d. Stacks

Recap

9. In which data structure, the element which is inserted first will be removed last (FILO principle)?

a. Linked lists
b. Arrays
c. Queues
d. Stacks

10. The data structure, in which the data items are physically not next to each other, but virtually in linear order is called a -----?

a. Arrays
b. Queues
c. Linked lists
d. Stacks

Recap

11. Based on organization of data, the classification of data structures are ?
    a. Linear and non-linear DS
    b. Primitive and non-primitive DS
    c. Contiguous and non-contiguous DS
    d. All the above

12. Based on arrangement of data elements in memory, the classification of data structures are
    a. Linear and non-linear DS
    b. Primitive and non-primitive DS
    c. Contiguous and non-contiguous DS
    d. All the above

Recap

13. Arrays and structures follow which category of data structures?
    a. Non-linear DS
    b. Primitive DS
    c. Contiguous DS
    d. All the above

14. Linked lists is an example of which data structures?
    a. Linear DS
    b. Non-primitive DS
    c. Non-contiguous DS
    d. All the above

Recap

15. A list which display the relationship of adjacency between elements is said to be?

a. Linear
b. Non-linear
c. Linked lists
d. Trees

Algorithm

Definition: An algorithm is a **finite sequence** of instructions, each of which has a **clear meaning** and can be performed with a **finite amount of effort** in a **finite length of time**.

Properties of an algorithm:

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

Practical Algorithm Design Issues

Choosing an efficient algorithm or data structure is an important aspect of the design process. There are three basic design goals that we should strive for in a program:

Design issues of an algorithm:

- Try to save time (Time Complexity)
- Try to save space (Space Complexity)
- Try to have face

Performance of a Program

The performance of a program is the amount of computer memory and time needed to run a program.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the Time Complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

- Instruction Space
- Data Space
- Environment Stack Space

Classification of Algorithms

- 1 (Constant running time)
- log n (logarithmic)
- n (linear)
- n log n
- n^2 (Quadratic)
- n^3 (Cubic)
- n^k (Polynomial)
- 2^n (Exponential)

If n is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Complexity of Algorithms

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data.
- Mostly, the storage space required by an algorithm is simply a multiple of the data size n. Complexity shall refer to the running time of the algorithm.
- The function f(n), gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data.
- The complexity function f(n) for certain cases are:
  - ✓ Best Case: Provides a lower bound on running time.
  - ✓ Average Case: Provides the expected running time.
  - ✓ Worst Case: Provides an upper bound on running time.

Asymptotic Notations

Run time of an algorithm is described as the function of input size n. Time Complexity is described ***asymptotically*** i.e. as the input size goes to infinity.

Time Complexity is expressed using the ***highest-order term*** in the expression for the exact running time. E.g. $f(n) = n^2 + 4n + 5$, then it is written as $\Theta(n^2)$

- Big-Oh(O) – Upper bound of a function
- Big-Omega(Ω) – Lower bound of a function
- Theta(Θ) – Average bound of a function

Big-Oh(O) Notations

$f(n) = O(g(n))$: there exist positive constants c and n_0 such that

$0 \le f(n) \le cg(n)$ for all $n \ge n_0$

g(n) is asymptotic upper bound for f(n).

$1 < \log n < n < n\log n < n^2 < n^3 \ldots < 2^n \ldots < n^n$

Example: f(n) = 2n + 3

$2n + 3 \le 10n$

Then f(n) = O(n)

$2n + 3 \le 7n \quad \forall\ n \ge 1$

Then f(n) = O(n)

$2n + 3 \le 2n + 3n \quad \forall\ n \ge 1$

$2n + 3 \le 5n$

Then f(n) = O(n)

f(n) = 2n + 3

$2n + 3 \le 2n^2 + 3n^2 \quad \forall\ n \ge 1$

$2n + 3 \le 5n^2$

Then f(n) = $O(n^2)$

$2n + 3 \le 5n^3 \quad \forall\ n \ge 1$

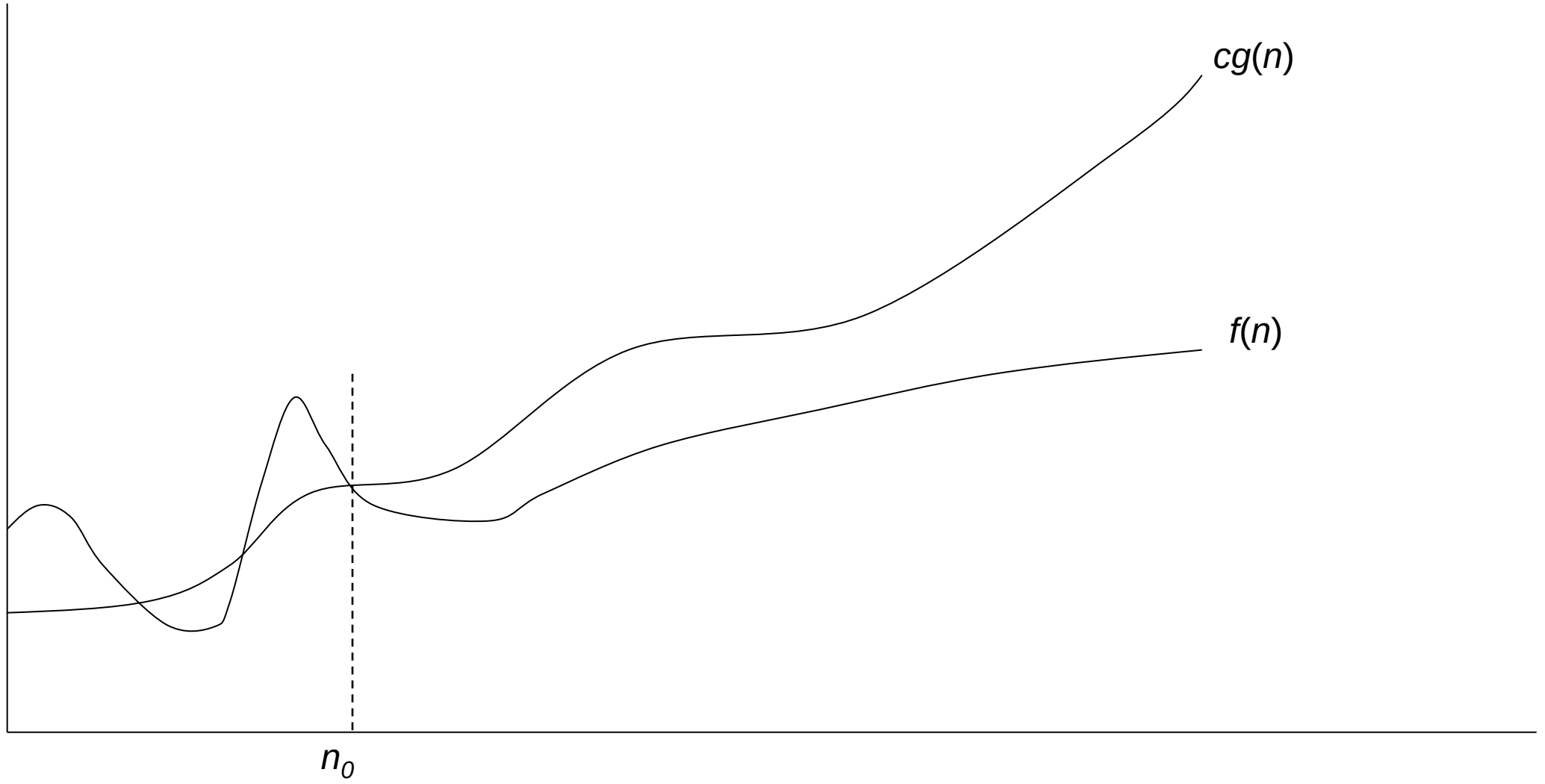
Then f(n) = $O(n^3)$

$2n + 3 \le 5n^5 \quad \forall\ n \ge 1$

Then f(n) = $O(n^5)$

f(n) = O(n)

Visualization of $O(g(n))$



Big-Omega(Ω) Notations

$f(n) = \Omega(g(n))$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \geq cg(n) \text{ for all } n \geq n_0$$

g(n) is asymptotic lower bound for f(n).

$1 < \log n < n < n\log n < n^2 < n^3 \ldots < 2^n \ldots < n^n$

Example: f(n) = 2n + 3

f(n) ≥ c*g(n)

2n + 3 ≥ 1*n ∀ n ≥ 1

Then f(n) = Ω(n) is true

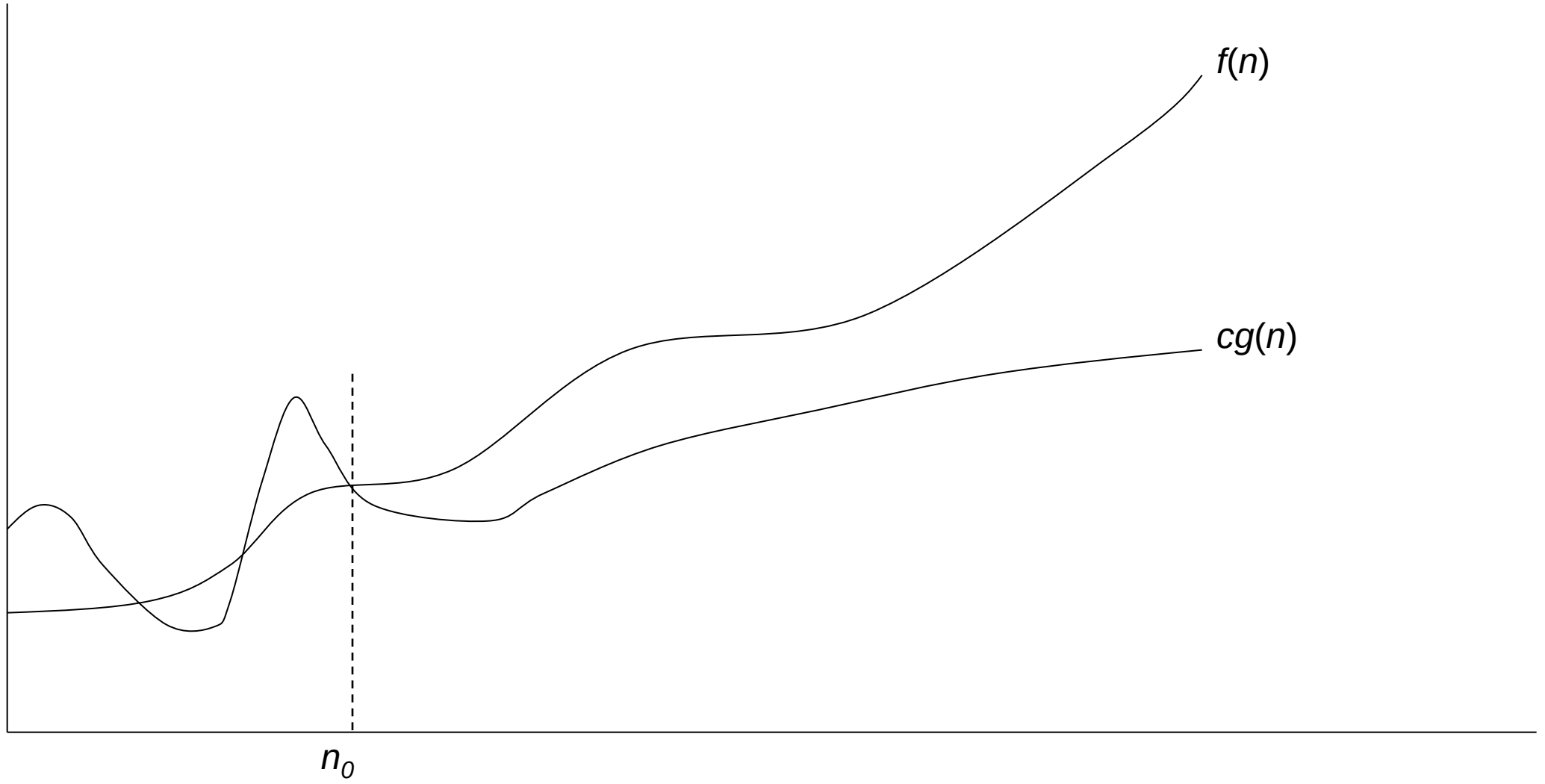
2n + 3 ≥ 1*log n ∀ n ≥ 1

Then f(n) = Ω(log n) is true

2n + 3 ≥ 1*n^2 ∀ n ≥ 1 is not satisfied

Then f(n) = Ω(n^2) is false

Visualization of $\Omega(g(n))$



Theta(Θ) Notations

$f(n) = \Theta(g(n))$: there exist positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)) \text{ AND } f(n) = \Omega(g(n))$$

g(n) is average or tight bound for f(n).

$1 < \log n < n < n\log n < n^2 < n^3 \ldots < 2^n \ldots < n^n$

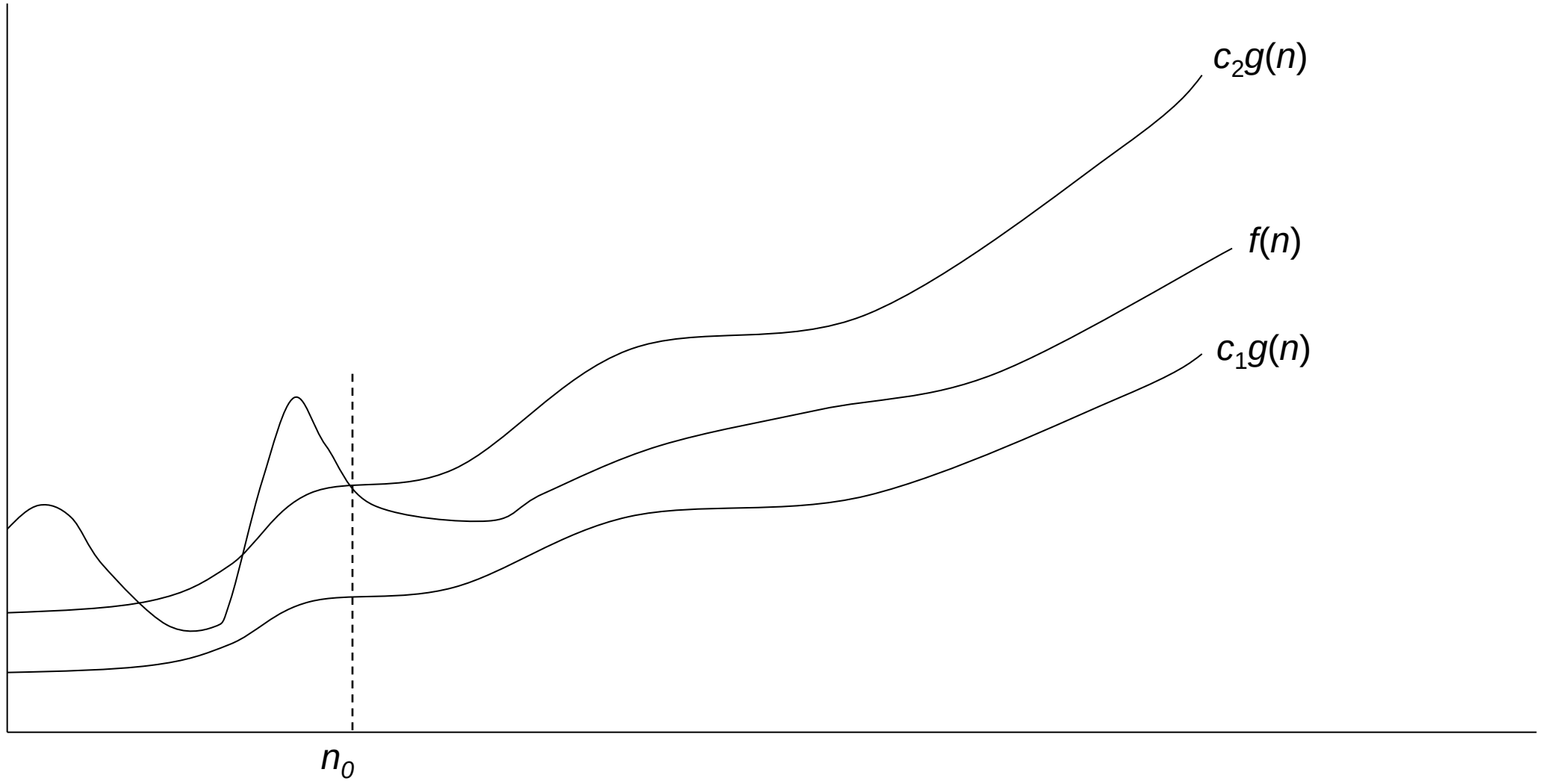
Example: f(n) = 2n + 3

$c_1 * g(n) \le f(n) \le c_2 * g(n)$

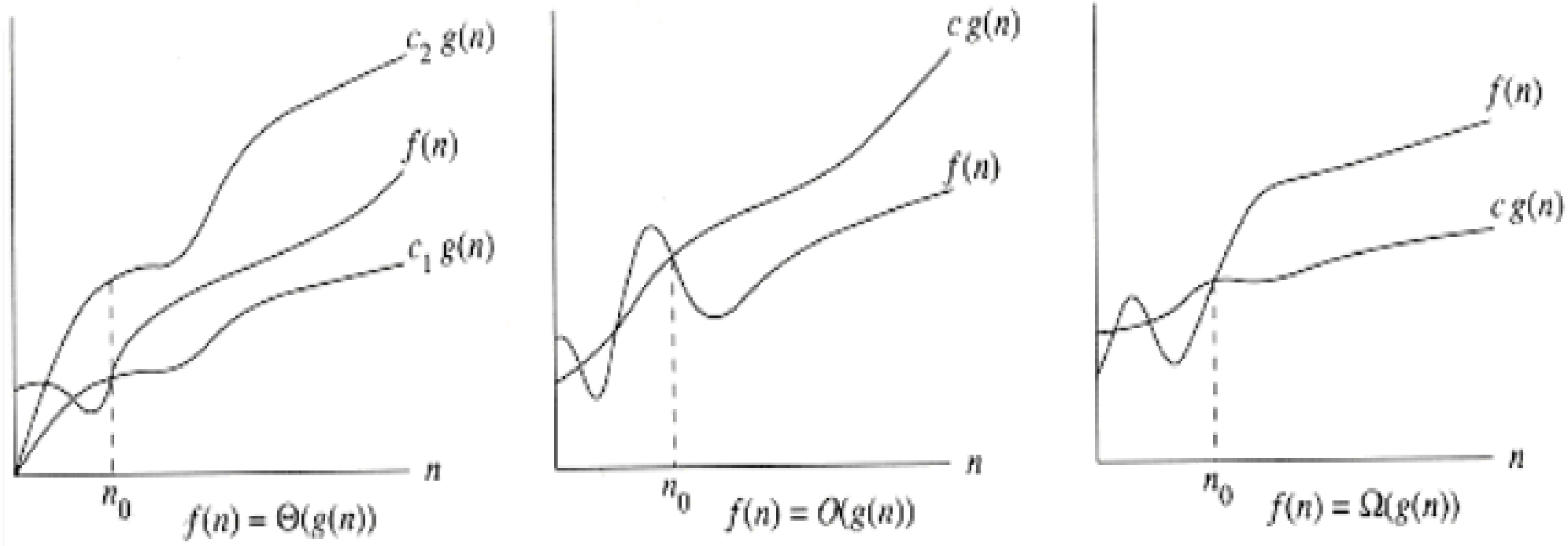
$1*n \le 2n + 3 \le 5*n \qquad \forall\ n \ge 1$

Then f(n) = Θ(n) is true

Visualization of $\Theta(g(n))$



Big-Oh (O), Omega (Ω) and Theta (Θ) Notations



More Examples on Asymptotic Notations

$1 < \log n < n < n\log n < n^2 < n^3 \ldots. < 2^n < 3^n < 4^n < \ldots < n^n$

Example: $f(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 \leq c * g(n)$

$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \quad \forall\ n \geq 1$

$2n^2 + 3n + 4 \leq 9n^2$

Then $f(n) = O(n^2)$ is true

Example: $f(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 \geq c * g(n)$

$2n^2 + 3n + 4 \geq 1 * n^2 \quad \forall\ n \geq 1$

Then $f(n) = \Omega(n^2)$ is true

Example: $f(n) = 2n^2 + 3n + 4$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$1 * n^2 \leq 2n^2 + 3n + 4 \leq 9n^2 \quad \forall\ n \geq 1$

Then $f(n) = \Theta(n^2)$ is true

More Examples on Asymptotic Notations

$$1 < \log n < n < n\log n < n^2 < n^3 \ldots. < 2^n < 3^n < 4^n < \ldots < n^n$$

Example: $f(n) = n^2 \log n + n$ represent the function f(n) using Big-Oh, Omega and Theta notation.

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

$$1 * n^2 \log n \leq n^2 \log n + n \leq 10 * n^2 \log n \qquad \forall\, n \geq 1$$

Then $f(n) = O(n^2 \log n)$ is true

$f(n) = \Omega(n^2 \log n)$ is true

$f(n) = \Theta(n^2 \log n)$ is true

More Examples on Asymptotic Notations

$$1 < \log n < n < n\log n < n^2 < n^3 \ldots. < 2^n < 3^n < 4^n < \ldots < n^n$$

Example: f(n) = n! represent the function f(n) using Big-Oh, Omega and Theta notation.

n! = n * (n-1) * (n-2) * (n-3) * ……..* 3 * 2 * 1

We can also write as 1 * 2 * 3 * …….* (n-2) * (n-1) * n

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$1 * 1 * \ldots\ldots * 1 \leq 1 * 2 * 3 * \ldots\ldots.* n \leq n * n * n * \ldots\ldots.* n \quad \forall\ n \geq 1$

$1 \leq n! \leq n^n$ so here g(n) is different in both sides.

Then $f(n) = O(n^n)$ is true and this is upper bound.

$f(n) = \Omega(1)$ is true and this is lower bound.

But we are unable to find a suitable place in between upper and lower bound. So we can't write theta notation for this function.

More Examples on Asymptotic Notations

$1 < \log n < n < n\log n < n^2 < n^3 \ldots. < 2^n < 3^n < 4^n < \ldots < n^n$

Example: f(n) = log n! represent the function f(n) using Big-Oh, Omega and Theta notation.

n! = n * (n-1) * (n-2) * (n-3) * ……* 3 * 2 * 1

We can also write as 1 * 2 * 3 * …….* (n-2) * (n-1) * n

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$\log(1 * 1 * \ldots\ldots * 1) \leq \log(1 * 2 * 3 * \ldots\ldots * n) \leq \log(n * n * \ldots.. * n) \quad \forall\, n \geq 1$

$1 \leq \log n! \leq \log n^n$ so here g(n) is different in both sides.

Then f(n) = O(log n^n) is true and this is upper bound.

f(n) = Ω(1) is true and this is lower bound.

Here we can't write theta notation for this function.

Note: Every function may not have an average or tight bound, in that case express the function using either upper or lower case.

Time Complexity Analysis of Loops

Example 1: for (i = 0; i < n; i++) ---------- n+1 times or n times

{

stmt; ---------- n times

}

Time Complexity ---------- O(n)

Example 2: for (i = 1; i > 0; i--) ---------- n+1 times or n times

{

stmt; ---------- n times

}

Time Complexity ---------- O(n)

Time Complexity Analysis of Loops

Example 3: for (i = 1; i < n; i=i+ 2)

```
{
    stmt;          ---------- n/2 times, so it's a function of f(n) = n/2
}
```

Time Complexity ---------- O(n)

Example 4: for (i = 1; i < n; i=i + 20)

```
{
    stmt;          ---------- n/20 times, so it's a function of f(n) = n/20
}
```

Time Complexity ---------- O(n)

Time Complexity Analysis of Loops

Example 5: for (i = 0; i < n; i++) ---------- n+1 times

{

for(j=0; j<n; j++) ---------- n * (n+1)

{

stmt; ---------- n * n times, so it's a function of $f(n) = n^2$

}

}

Time Complexity ---------- $O(n^2)$

Time Complexity Analysis of Loops

```
Example 6: for (i = 0; i < n; i++)
                {
                    for(j=0; j<i; j++)
                        {
                            stmt;
                         }
                }
```

1+2+3+4+……+n = (n(n+1)) / 2 times

$f(n) = (n^2+1)/2$

Time Complexity $O(n^2)$

i	j	no of times stmt executed
0	false	stmt not executed
1	0	1 time
	1	false
2	0	2 times
	1	
	2	false
3	0	3 times
	1	
	2	
	3	false
……		
n	0	n times
	1	
	2	
	…..	
	n-1	
	n	false

Time Complexity Analysis of Loops

Example 7:

```
p=0;
for (i = 1;p <= n; i++)
{
        p = p+i;
}
```

Assume when p > n then the loop will stop.

Since p = (k(k+1))/2

(k(k+1))/2 > n

Assume $k^2 > n$

$k > \sqrt{n}$

Time Complexity f(n) = $O(\sqrt{n})$

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3 = 6
4	1+2+3+4 = 10
5	1+2+3+4+5 = 15
……	
K	1+2+3+4+5+…..+K

Time Complexity Analysis of Loops

Example 8:

```
for (i = 1; i < n; i = i*2)
{
        stmt;
}
```

Assume when i > n then the loop will stop.

Since $i = 2^k$

$2^k >= n$

Take $2^k = n$

$k = \log_2 n$

Time Complexity $f(n) = O(\log_2 n)$

i
1
$1 \times 2 = 2$
$2 \times 2 = 2^2$
$2^2 \times 2 = 2^3$
$2^3 \times 2 = 2^4$
……
$2^{k-1} \times 2 = 2^k$

Time Complexity Analysis of Loops

Example 9:

```
for (i = n; i >= 1; i = i/2)
{
        stmt;
}
```

Assume when i < 1 then the loop will stop.

Since $i = n / 2^k$

$n / 2^k < 1$

So $n = 2^k$

$k = \log_2 n$

Time Complexity $f(n) = O(\log_2 n)$

i
n
$n / 2$
$n / 2^2$
$n / 2^3$
$n / 2^4$
……
$n / 2^k$

Time Complexity Analysis of Loops

Example 10:

```
for (i = 0; i * i < n; i++)
{
        stmt;
}
```

Till i * i < n the loop will execute,

when i * i >= n then the loop will stop.

Since $i^2 >= n$

So $i^2 = n$

$i = \sqrt{n}$

Time Complexity $f(n) = O(\sqrt{n})$

Time Complexity Analysis of Loops

Example 11: Independent loops

```
for (i = 0; i < n; i++)
{
        stmt;           ---------------- n times
}
for (j = 0; j < n; j++)
{
        stmt;           ---------------- n times
}
```

Total 2n times both the loops will run.

Time Complexity f(n) = O(n)

Time Complexity Analysis of Loops

Example 12: Dependent loops

```
p = 0;
for (i = 1; i < n; i = i*2)
{
        p++;            ---------- log n times
}
for (j = 1; j < p; j = j*2)
{
        stmt;           ---------- log p times = log log n times
}
```

Time Complexity f(n) = O(log log n)

Time Complexity Analysis of Loops

Example 13: Nested loops

```
for (i = 0; i < n; i ++)    --------- n+1 times or n times
{
   for (j = 1; j < n; j = j*2)   ------ n log n times
   {
          stmt;                 ---------------- n log n times
   }
}
```

Total both the loops will run 2nlog n + n times

Time Complexity f(n) = O(nlog n)

Recap

1. The Big-Oh notation for $f(x) = 5 \log x$ is ?
   a. 1
   b. x
   c. x^2
   d. X^3
   Ans: $5 \log x \leq x$ then $f(x) = O(x)$

2. The Big-Omega notation for $f(x) = 2x^4 + x^2 - 4$?
   a. x^2
   b. x^3
   c. x
   d. X^4
   Ans: $2x^4 + x^2 - 4 \geq x^4$ then $f(x) = \Omega(x^4)$

Searching

Definition: Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.

Searching Techniques:

- Linear or Sequential Search
- Binary Search
- Fibonacci Search
- Interpolation Search

Linear or Sequential Search

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Time Complexity of Linear Search algorithm:

- Suppose there are n elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.
- The time complexity of linear search
  - Best case: O(1)
  - Average case: O(n/2)
  - Worst case: O(n)

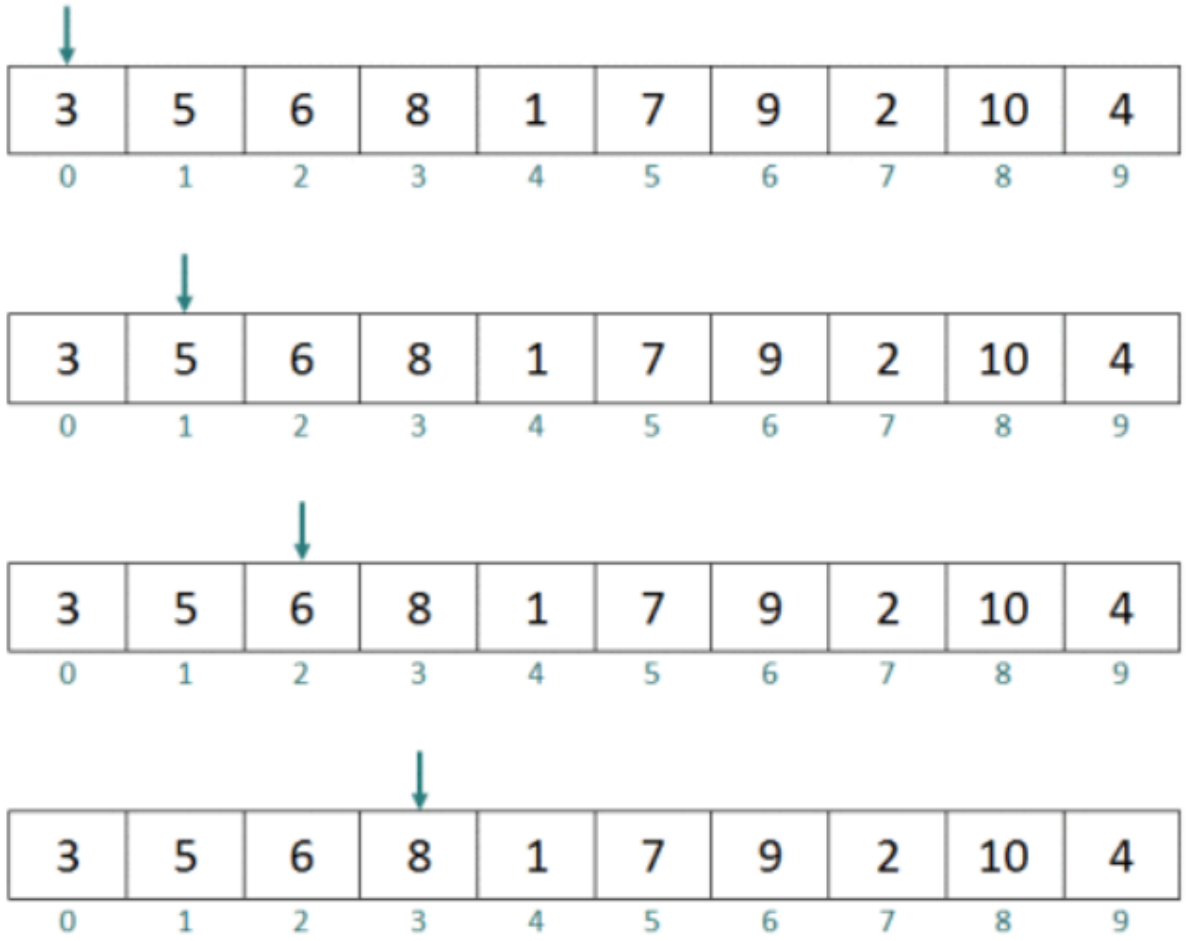
Linear Search Algorithm

Let array a[n] stores n elements. Determine whether element x is present or not.

```
linear_search(a[n], x)
{
        index = 0; flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf("Data found at %d position", index);
        else
                printf("data not found");
}
```

Linear Search Example

Array = { 3, 5, 6, 8, 1, 7, 9, 2, 10, 4 } Find the position of 8.



Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of $O(\log_2 n)$.
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.
- The time complexity of Binary search
  - Best case: $O(1)$
  - Average case: $O(\log_2 n)$
  - Worst case: $O(\log_2 n)$

Binary Search Example

index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4 then
Low =1, high = 12, mid = 6, check 20
Low = 1, high = 5, mid = 3, check 8
Low = 1, high = 2, mid = 1, check 4, Data Found (3 comparisons)

If we are searching for x = 77
Low = 1, high = 12, mid = 6, check 20
Low= 7, high = 12, mid = 9, check 39
Low = 10, high = 12, mid = 11, check 54
Low = 12, high = 12, mid = 12, check 77, Data Found

Time Complexity of Binary search : O(log n) (4 comparisons)

Binary Search Algorithm

```
binary_search(a[], n, x)
{
        low = 1; high = n;
        while (low < high) do
        {
                mid = (low + high)/2;
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else
                        return mid;
        }
        return 0;
}
```

Sorting Techniques

Definition: Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose.

For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order.

There are a number of many sorting techniques such as:

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort
- Radix sort

There are two types of sorting techniques:

- Internal sorting
- External sorting

Bubble Sort or Exchange Sort

➢Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

➢This sorting technique is also known as exchange sort, which arranges values by iterating over the list several times and in each iteration the larger value gets bubble up to the end of the list.

➢Consider the array x[n] which is stored in memory as shown below:

Initial Array or List

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Pass 1:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

Bubble Sort or Exchange Sort

Pass 2:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

Pass 3:

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4:

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22 33		

Pass 5: The array will be sorted

Bubble Sort and Modified (Optimized) Bubble Sort Algorithm

```
begin BubbleSort(list)

   for all elements of list
       if list[i] > list[i+1]
           swap(list[i], list[i+1])
       end if
   end for

   return list

end BubbleSort
```

```
void bubble(int a[], int n){
        for(int i=0; i<n; i++){
                int swaps=0;
                for(int j=0; j<n-i-1; j++){
                        if(a[j]>a[j+1]){
                                int t=a[j];
                                a[j]=a[j+1];
                                a[j+1]=t;
                                swaps++;
                        }
                }
                if(swaps==0)
                        break;
        }
}
```

Time Complexity of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
Sum = n(n-1)/2
i.e. $O(n^2)$

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = $n^2 - 2n + 1$, which is $O(n^2)$.

Therefore bubble sort is very inefficient when there are more elements to sorting.

- Worst Case Time Complexity: $O(n^2)$ upper bound
- Best Case Time Complexity: $\Omega(n)$ lower bound (when the list is already sorted)
- Average Time Complexity: $O(n^2)$ average bound
- Space Complexity: O(1) because only a single additional memory space is required i.e. for temp variable.

Recap

1. Where is linear searching used?
a) When the list has only a few elements
b) When performing a single search in an unordered list
c) Used all the time
d) When the list has only a few elements and When performing a single search in an unordered list

2. What is the best case for linear search?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

3. What is the worst case for linear search?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

Recap

4. What is the best case and worst case complexity of ordered linear search?
a) O(nlogn), O(logn)
b) O(logn), O(nlogn)
c) O(n), O(1)
d) O(1), O(n)

5. Which of the following is a disadvantage of linear search?
a) Requires more space
b) Greater time complexities compared to other searching algorithms
c) Not easy to understand
d) Not easy to implement

6. What is the advantage of recursive approach than an iterative approach?
a) Consumes less memory
b) Less code and easy to implement
c) Consumes more memory
d) More code has to be written

Recap

7. Given an input arr = {2,5,7,99,899}; key = 899; What is the level of recursion?
a) 5
b) 2
c) 3
d) 4

8. Given an array arr = {45,77,89,90,94,99,100} and key = 99; what are the mid values(corresponding array elements) in the first and second levels of recursion?
a) 90 and 99
b) 90 and 94
c) 89 and 99
d) 89 and 94

9. What is the worst case complexity of binary search using recursion?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(n2)

Recap

10. What is the average case time complexity of binary search using recursion?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(n2)

11. Which of the following is not an application of binary search?
a) To find the lower/upper bound in an ordered sequence
b) Union of intervals
c) Debugging
d) To search in unordered list

12. Binary Search can be categorized into which of the following?
a) Brute Force technique
b) Divide and conquer
c) Greedy algorithm
d) Dynamic programming

Recap

13. Given an array arr = {5,6,77,88,99} and key = 88; How many iterations are done until the element is found in binary search?
a) 1
b) 3
c) 4
d) 2

14. What is the time complexity of binary search with iteration?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(n2)

15. Is there any difference in the speed of execution between linear serach(recursive) vs linear search(lterative)?
a) Both execute at same speed
b) Linear search(recursive) is faster
c) Linear search(Iterative) is faster
d) Cant be said

Recap

16. Is the space consumed by the linear search(recursive) and linear search(iterative) same?
a) No, recursive algorithm consumes more space
b) No, recursive algorithm consumes less space
c) Yes
d) Nothing can be said

17. What is the worst case runtime of linear search(recursive) algorithm?
a) O(n)
b) O(logn)
c) O(n2)
d) O(nx)

18. Linear search(recursive) algorithm used in _____
a) When the size of the dataset is low
b) When the size of the dataset is large
c) When the dataset is unordered
d) Never used

Recap

19. Can linear search recursive algorithm and binary search recursive algorithm be performed on an unordered list?
a) Binary search can’t be used
b) Linear search can’t be used
c) Both cannot be used
d) Both can be used

20. What is an external sorting algorithm?
a) Algorithm that uses tape or disk during the sort
b) Algorithm that uses main memory during the sort
c) Algorithm that involves swapping
d) Algorithm that are considered ‘in place’

21. What is an internal sorting algorithm?
a) Algorithm that uses tape or disk during the sort
b) Algorithm that uses main memory during the sort
c) Algorithm that involves swapping
d) Algorithm that are considered ‘in place’

Recap

22. What is the worst case complexity of bubble sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

23. What is the average case complexity of bubble sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

24. Bubble sort is also known as ----------

Selection Sort

- Selection sort is an in-place comparison sorting algorithm.
- The algorithm divides the input list into two parts:
  - A sorted sub-list of items which is built up from left to right at the front (left) of the list and
  - A sub-list of the remaining unsorted items that occupy the rest of the list.
- Initially, the sorted sub-list is empty and the unsorted sub-list is the entire input list.
- The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub-list, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sub-list boundaries one element to the right.

Selection Sort

Let us consider the following example with 9 elements to analyze selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Selection Sort Example

Sorted sublist	Unsorted sublist	Least element in unsorted list
()	(11, 25, 12, 22, 64)	11
(11)	(25, 12, 22, 64)	12
(11, 12)	(25, 22, 64)	22
(11, 12, 22)	(25, 64)	25
(11, 12, 22, 25)	(64)	64
(11, 12, 22, 25, 64)	()	

Time Complexity of Selection Sort

- The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort.
- One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, n − 1 in the worst case.

Selection sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n)$ swaps
Best-case performance	$O(n^2)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons, $O(n)$ swaps
Worst-case space complexity	O(1) auxiliary

Selection Sort Algorithm

```
FindMinIndex(Arr[], start, end)
        min_index = start

        FOR i from (start + 1) to end:
            IF Arr[i] < Arr[min_index]:
                min_index = i
            END of IF
        END of FOR

  Return min_index
```

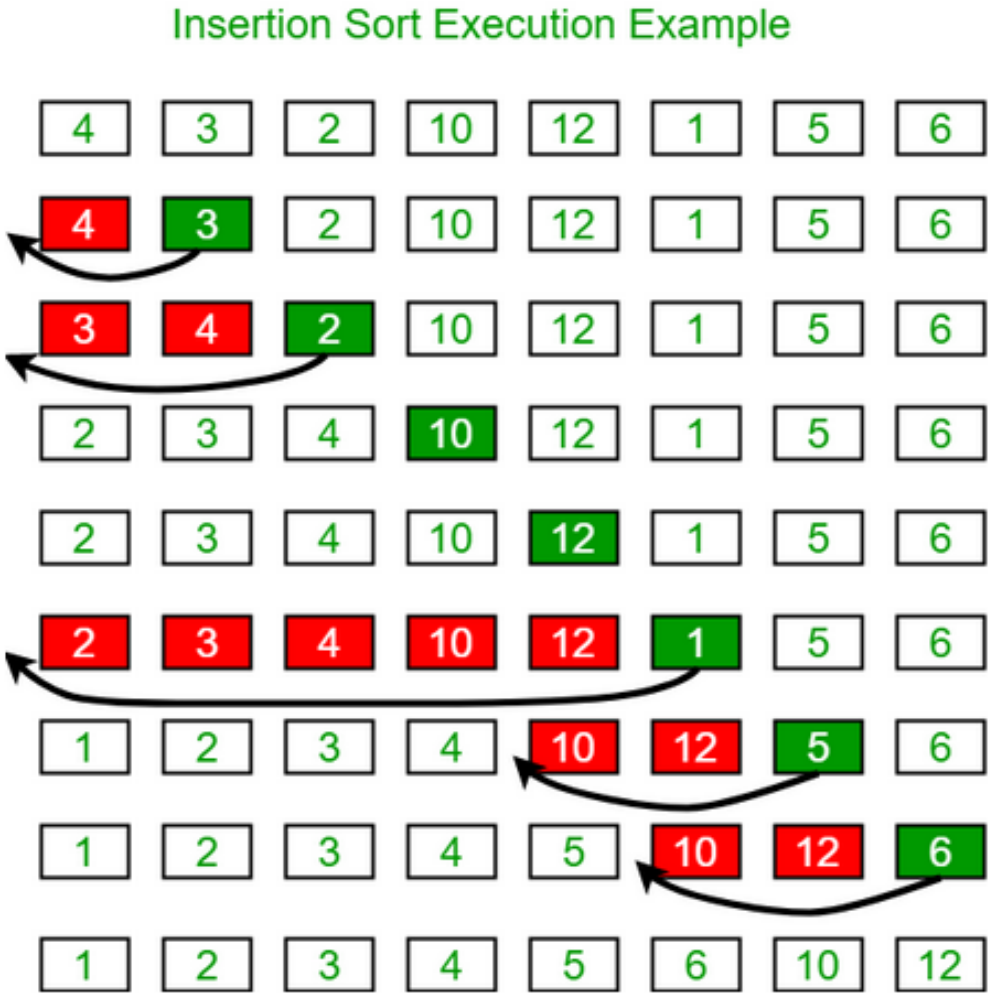
Insertion Sort

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.
- An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Example:

- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.

Insertion Sort



Advantages of Insertion Sort

- Efficient for (quite) small data sets, much like other quadratic sorting algorithms.
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(kn) when each element in the input is no more than k places away from its sorted position.
- Stable; i.e., does not change the relative order of elements with equal keys.
- In-place; i.e., only requires a constant amount O(1) of additional memory space.
- Online; i.e., can sort a list as it receives it.

Time Complexity of Insertion Sort

➢ The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

➢The simplest worst case input is an array sorted in reverse order. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., $O(n^2)$).

➢The average case is also quadratic ($O(n^2)$), which makes insertion sort impractical for sorting large arrays.

➢ However, insertion sort is one of the fastest algorithms for sorting very small arrays.

Time Complexity of Insertion Sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Insertion Sort Algorithm

```
INSERTION-SORT(A)
   for i = 1 to n
         key ← A [i]
         j ← i – 1
          while j > = 0 and A[j] > key
                  A[j+1] ← A[j]
                  j ← j – 1
         End while
         A[j+1] ← key
  End for
```

Comparison of Time Complexities of Various Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)

Recap

25. How many passes does an insertion sort algorithm consist of?
a) N
b) N-1
c) N+1
d) N^2

26. What is the average case running time of an insertion sort algorithm?
a) O(N)
b) O(N log N)
c) O(log N)
d) $O(N^2)$

27. Any algorithm that sorts by exchanging adjacent elements require O(N2) on average.
a) True
b) False

Recap

28. What is the running time of an insertion sort algorithm if the input is pre-sorted?
a) O(N2)
b) O(N log N)
c) O(N)
d) O(M log N)

29. What will be the number of passes to sort the elements using insertion sort?
14, 12,16, 6, 3, 10
a) 6
b) 5
c) 7
d) 1

30. Which of the following real time examples is based on insertion sort?
a) arranging a pack of playing cards
b) database scenarios and distributes scenarios
c) arranging books on a library shelf
d) real-time systems

Recap

31. Which of the following options contain the correct feature of an insertion sort algorithm?
a) anti-adaptive
b) dependable
c) stable, not in-place
d) stable, adaptive

32. Which of the following sorting algorithms is the fastest for sorting small arrays?
a) Quick sort
b) Insertion sort
c) Shell sort
d) Heap sort

33. For the best case input, the running time of an insertion sort algorithm is?
a) Linear
b) Binary
c) Quadratic
d) Depends on the input

Recap

34. What is an in-place sorting algorithm?
a) It needs O(1) or O(logn) memory to create auxiliary locations
b) The input is already sorted and in-place
c) It requires additional storage
d) It requires additional space

35. What is the worst case complexity of selection sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

36. What is the advantage of selection sort over other sorting techniques?
a) It requires no additional storage space
b) It is scalable
c) It works best for inputs which are already sorted
d) It is faster than any other sorting technique

Recap

37. What is the disadvantage of selection sort?
a) It requires auxiliary memory
b) It is not scalable
c) It can be used for small keys
d) It takes linear time to sort the elements

38. What is the best case complexity of selection sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

39. The given array is arr = {3,4,5,2,1}. The number of iterations in bubble sort and selection sort respectively are,
a) 5 and 4
b) 4 and 5
c) 2 and 4
d) 2 and 5

Recap

40. The complexity of the sorting algorithm measures the …… as a function of the number n of items to be sorter.

A. average time

B. running time

C. average-case complexity

D. case-complexity

Module – 2
Linear Data Structures

Contents

- Stacks: Primitive operations
- Implementation of stacks using Arrays
- Applications of stacks arithmetic expression conversion and evaluation
- Queues: Primitive operations; Implementation of queues using Arrays
- Applications of linear queue
- Circular queue
- Double ended queue (deque)

Stacks

- A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.

- The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

- The basic implementation of a stack is also called a LIFO (Last In First Out).

- There are basically three operations that can be performed on stacks. They are

  1) inserting an item into a stack (push).

  2) deleting an item from the stack (pop).

  3) displaying the contents of the stack (peek or top).

Basic Stack Operations (Stack ADT)

PUSH: It is the term used to insert an element into a stack.

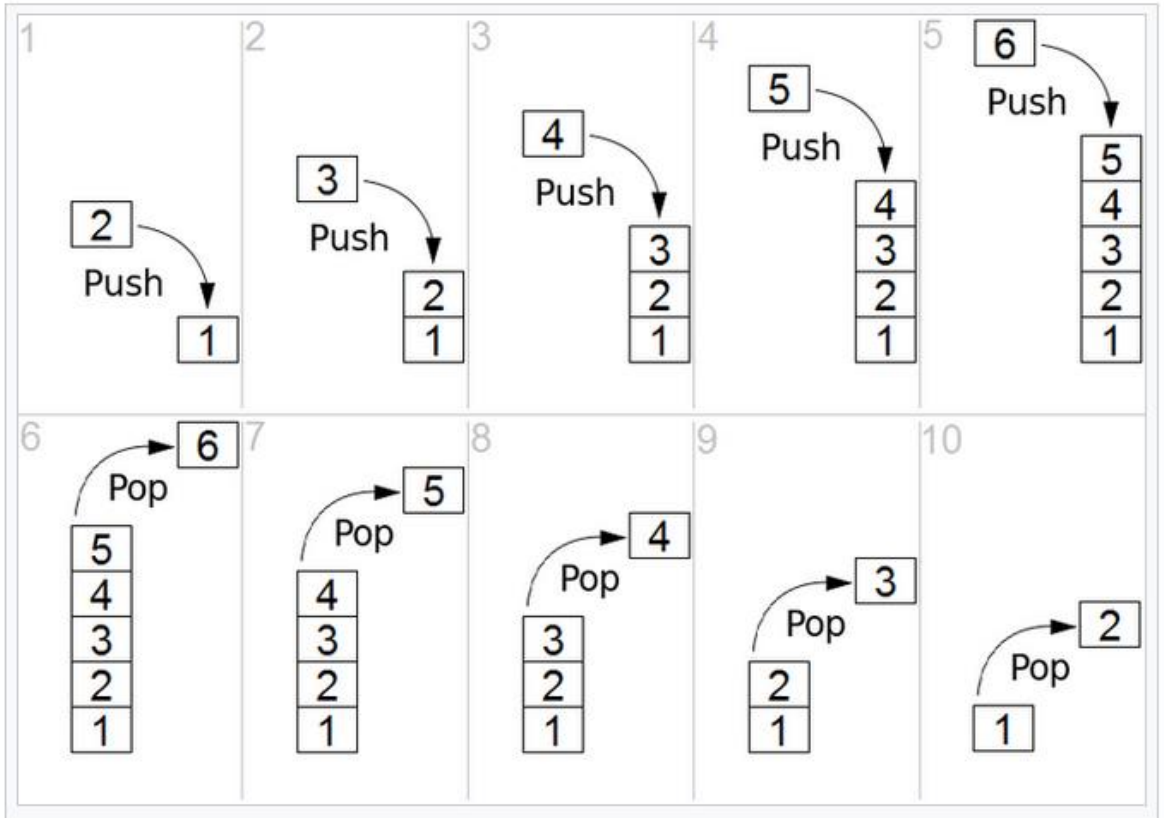
POP: It is the term used to delete an element from a stack.

Additional Operations:

Display: It displays all the elements in the stack.

Traverse: Visit each element of the stack from top to bottom or vice versa.

Search: Check whether a specific element is present in the stack or not.



Standard Error Messages in Stack

- Two standard error messages of stack are
  - **Stack Overflow**: If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition.
  - **Stack Underflow**: If we attempt to remove elements beyond the base of the stack, we will encounter a *stack underflow* condition.

Stack Operations

- **PUSH (STACK, TOP, MAXSTR, ITEM):** This procedure pushes an ITEM onto a stack
  1. If TOP = MAXSIZE, then Print: OVERFLOW, and Return.
  2. Set TOP := TOP + 1 [Increases TOP by 1]
  3. Set STACK [TOP] := ITEM. [Insert ITEM in TOP position]
  4. Return

- **POP (STACK, TOP, ITEM):** This procedure deletes the top element of STACK and assign it to the variable ITEM
  1. If TOP = 0, then Print: UNDERFLOW, and Return.
  2. Set ITEM := STACK[TOP]
  3. Set TOP := TOP - 1 [Decreases TOP by 1]
  4. Return

Stack Implementation

- There are many ways of implementing stack ADT, below are the commonly used methods:

➢ Static array/list based implementation

➢ Dynamic array/list based implementation

  ➢ Linked lists implementation

Dynamic implementation of stack

```
stack = []       #stack is a list, stack is an empty list

# append() function to push element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() fucntion to pop element from stack in LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('Stack after elements are popped:')
print(stack)   #Display the elements of stack
```

Static implementation of stack

```
#Stack implementation using list
top=0
mymax=5
def createStack():
    stack=[]       #stack is a list
    return stack

def isEmpty(stack):
    return len(stack)==0

def Push(stack,item):
    stack.append(item)
    print("Pushed to stack", item)

def Pop(stack):
    if isEmpty(stack):
        return "stack underflow"
    return stack.pop()
```

```
#create a stack object
stack=createStack()
while True:
    print("1.Push")
    print("2.Pop")
    print("3.Display")
    print("4.Quit")
    ch=int(input("Enter your choice:"))
    if ch==1:
        if top < mymax:
            item=input("Enter any elements:")
            Push(stack, item)
            top +=1
        else:
            print("Stack overflow")
    elif ch==2:
        print(Pop(stack))
    elif ch==3:
        print(stack)
    else:
        break
```

Applications of Stack

1. Stack is used by compilers to check for balancing of parentheses.

2. Stack is used to evaluate a prefix and postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor's stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

6. Depth first search (DFS) uses a stack data structure to find an element from a graph.

7. Page-visited history in a web browser.

8. Undo sequence in a text editor.

9. Matching tags in HTML and XML.

Time Complexity of Stack

Let *n* be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Algebraic Expression Conversion

Algebraic Expression Conversion

1. Infix to Postfix / Prefix

2. Prefix to Infix / Postfix

3. Postfix to Infix / Prefix

Infix Expression: (A + B) / (C - D)

Postfix Expression: A B + C D - /

Prefix Expression: / + A B - C D

Conversion of Infix to Postfix Expression

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.

2. a) If the scanned symbol is left parenthesis, push it onto the stack.

b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Conversion of Infix to Postfix Expression

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

Symbol	Postfix string	Stack	Remarks
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	-	
-	A B C * +	-	
D /	A B C * + D A B C * + D	- - /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion of Infix to Postfix Expression

Infix to Postfix Conversion

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion of Infix to Postfix Expression

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion of Infix to Prefix Expression

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion of Infix to Prefix Expression

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion of Postfix to Infix Expression

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) – ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) – ((D/(E^F))*G)) H	Push H
*	A (((B*C) – ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

Conversion of Postfix to Prefix Expression

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*-*BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

Conversion of Prefix to Infix Expression

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the string.
*	(((B*C)-((D/(E^F))*G))*H)	Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H) A	Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))	Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.	

Conversion of Prefix to Postfix Expression

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G EF^	Pop two operands and place the operator after the operands and push the string.
D	H G EF^ D	Push D
/	H G DEF^/	Pop two operands and place the operator after the operands and push the string.
*	H DEF^/G*	Pop two operands and place the operator after the operands and push the string.
C	H DEF^/G* C	Push C
B	H DEF^/G* C B	Push B
*	H DEF^/G* BC*	Pop two operands and place the operator after the operands and push the string.
-	H BC*DEF^/G*-	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H*	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* A	Push A
+	ABC*DEF^/G*-H*+	Pop two operands and place the operator after the operands and push the string.
End of string	The input is now empty. The string formed is postfix.	

Evaluation of Postfix Expression

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	**288**	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Important Links

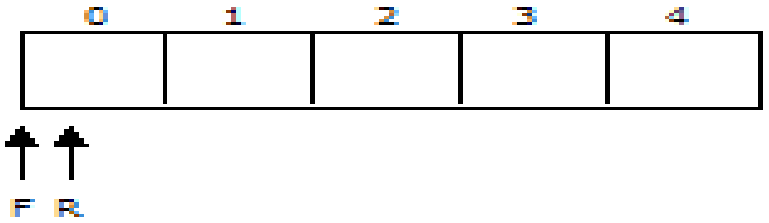
1. https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues
2. https://www.docdroid.net/ZPfHmS5/data-structures-and-algorithms-narasimha-karumanchi-pdf
3. https://www.geeksforgeeks.org/stack-data-structure/
4. https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/tutorial/
5. https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Queue

- A queue is a data structure where items are inserted at one end called the rear and deleted at the other end called the front.
- Another name for a queue is a “FIFO” or “First-in-first-out” list.
- Operations of a Queue:
  - enqueue: which inserts an element at the end of the queue.
  - dequeue: which deletes an element at the front of the queue.

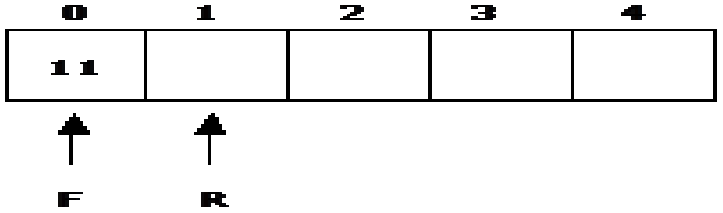
Representation of Queue

Initially the queue is empty.



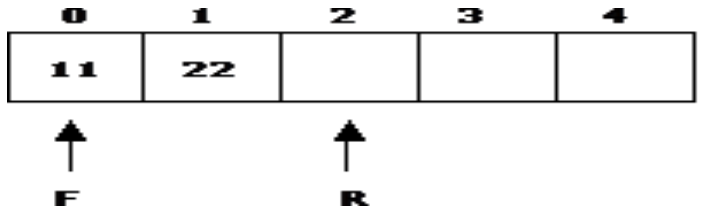
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



REAR = REAR + 1 = 1
FRONT = 0

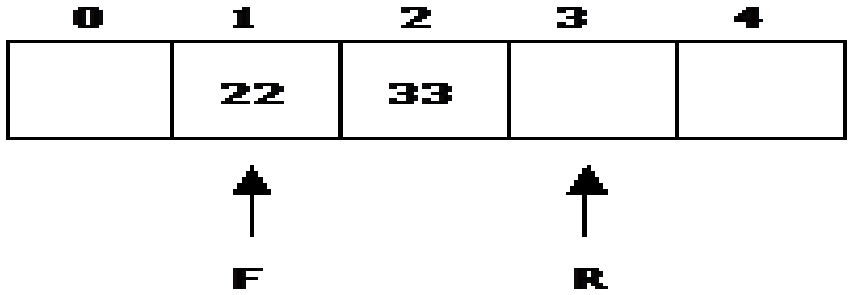
Next, insert 22 to the queue. Then the queue status is:



REAR = REAR + 1 = 2
FRONT = 0

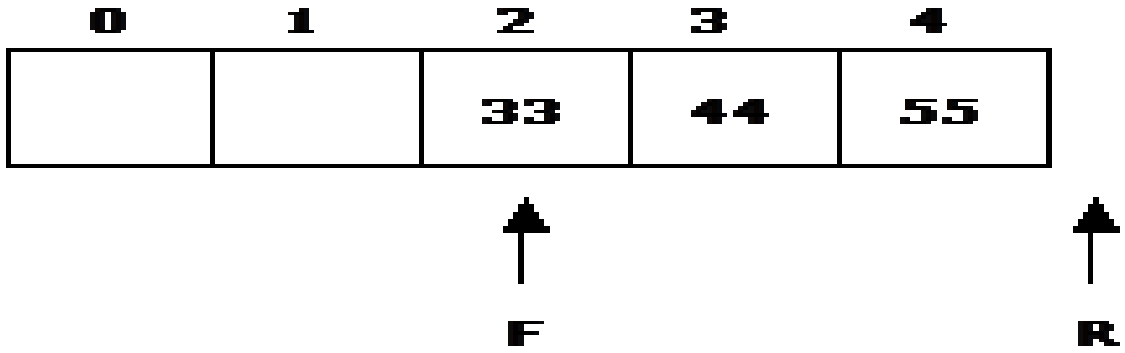
Representation of Queue

Now, delete an element 11.



REAR = 3
FRONT = FRONT + 1 = 1

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as it signals queue is full. The queue status is as follows:



REAR = 5
FRONT = 2

Queue Operations using Array

- Various operations of Queue are:
  - **insertQ():** inserts an element at the end of queue Q.
  - **deleteQ():** deletes the first element of Q.
  - **displayQ():** displays the elements in the queue.
- There are two problems associated with linear queue. They are:
  - **Time consuming**: linear time to be spent in shifting the elements to the beginning of the queue.
  - **Signaling queue full**: even if the queue is having vacant position.

Applications of Queue

- It is used to schedule the jobs to be processed by the CPU.
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue

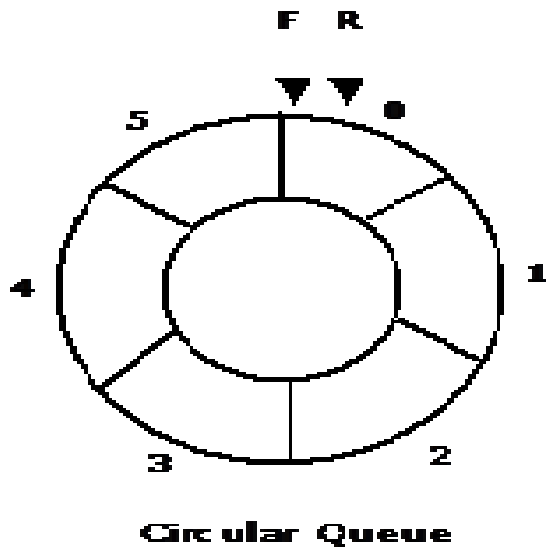
- A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.
- Suppose if we have a Queue of n elements then after adding the element at the last index i.e. $(n-1)^{th}$, as queue is starting with 0 index, the next element will be inserted at the very first location of the queue which was not possible in the simple linear queue.

Circular Queue Operations

- The Basic Operations of a circular queue are
  - **InsertionCQ**: Inserting an element into a circular queue results in Rear = (Rear + 1) % MAX, where MAX is the maximum size of the array.
  - **DeletionCQ** : Deleting an element from a circular queue results in Front = (Front + 1) % MAX, where MAX is the maximum size of the array.
  - **TraversCQ**: Displaying the elements of a circular Queue.
- Circular Queue Empty: Front=Rear=0.

Circular Queue Representation using Arrays

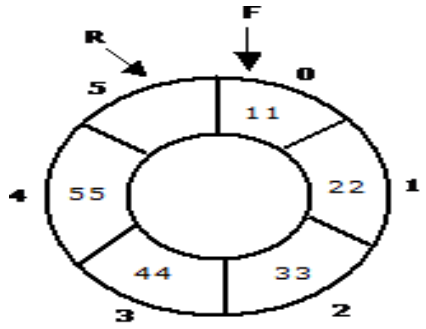
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

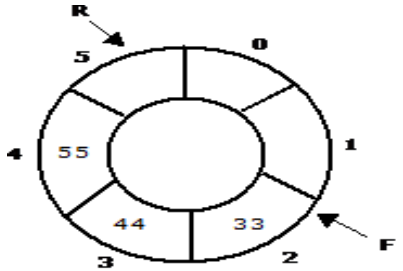
Insertion and Deletion Operations



Circular Queue

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

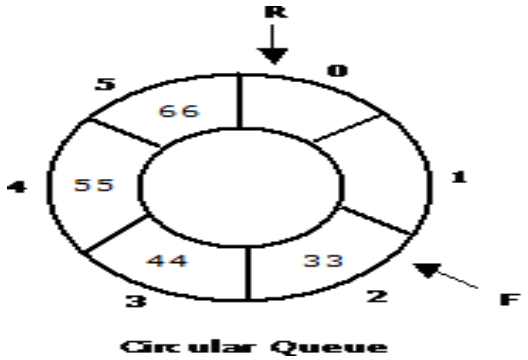
Now, delete two elements 11, 22 from the circular queue. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

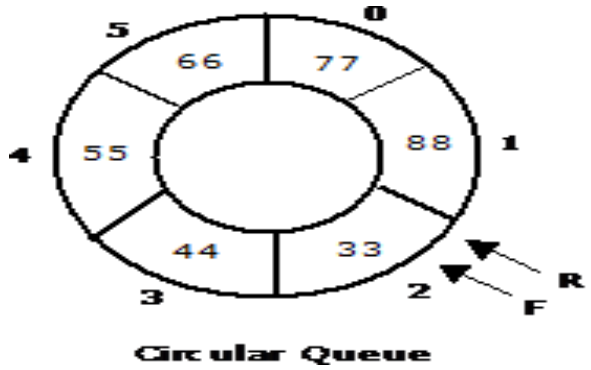
Insertion and Deletion Operations



Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

Again, insert 77 and 88 to the circular queue. The status of the Circular queue is:



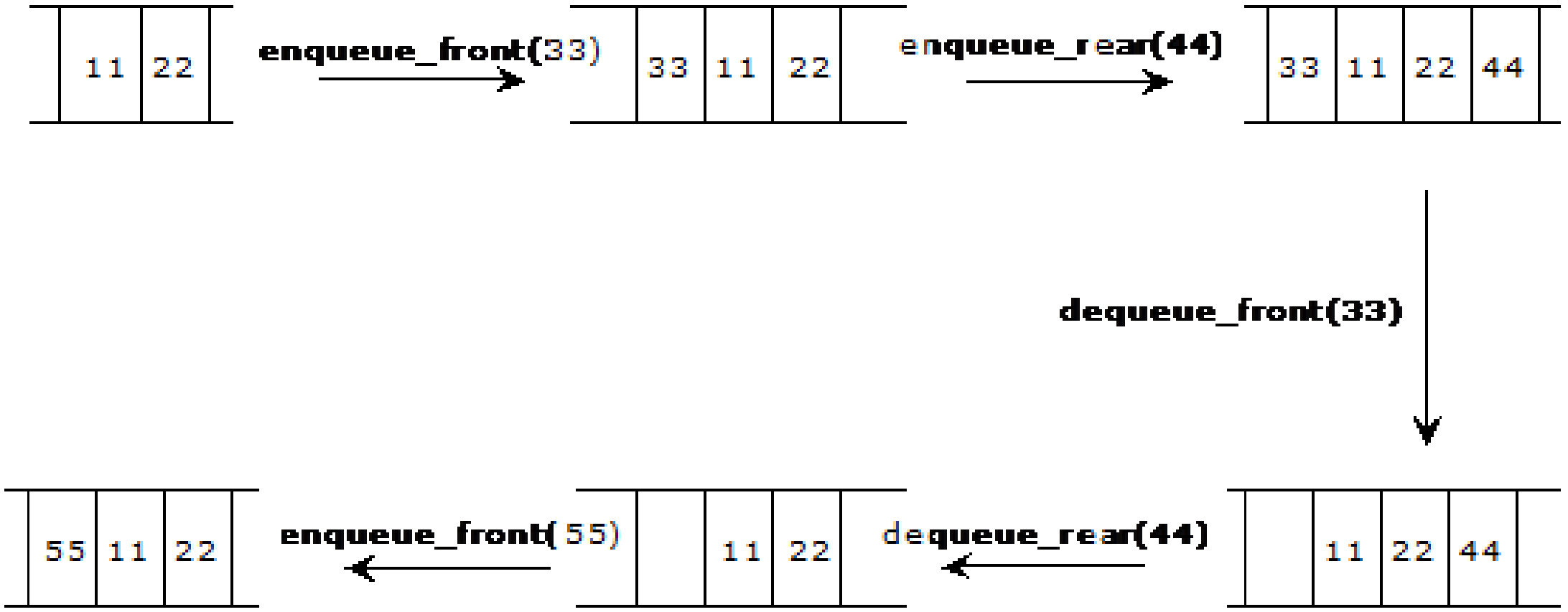
Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Double Ended Queue (DEQUE)

- It is a special queue like data structure that supports insertion and deletion at both the front and the rear of the queue.
- Such an extension of a queue is called a **double-ended queue, or deque,** which is usually pronounced **"deck"** to avoid confusion with the dequeue method of the regular queue, which is pronounced like the abbreviation **"D.Q."**
- It is also often called a **head-tail linked list**.

DEQUE Representation using Arrays



Types of DEQUE

- There are two variations of deque. They are:
  - Input restricted deque (IRD)
  - Output restricted deque (ORD)
- An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.
- An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

Priority Queue

- A priority queue is a collection of elements that each element has been assigned a priority and such that order in which elements are deleted and processed comes from the following riles:
  - An element of higher priority is processed before any element of lower priority.
  - Two element with the same priority are processed according to the order in which they were added to the queue.

Priority Queue operations and Usage

- Inserting new elements.
- Removing the largest or smallest element.
- Priority Queue Usages are:

  Simulations: Events are ordered by the time at which they should be executed.

  Job scheduling in computer systems: Higher priority jobs should be executed first.

  Constraint systems: Higher priority constraints should be satisfied before lower priority constraints.

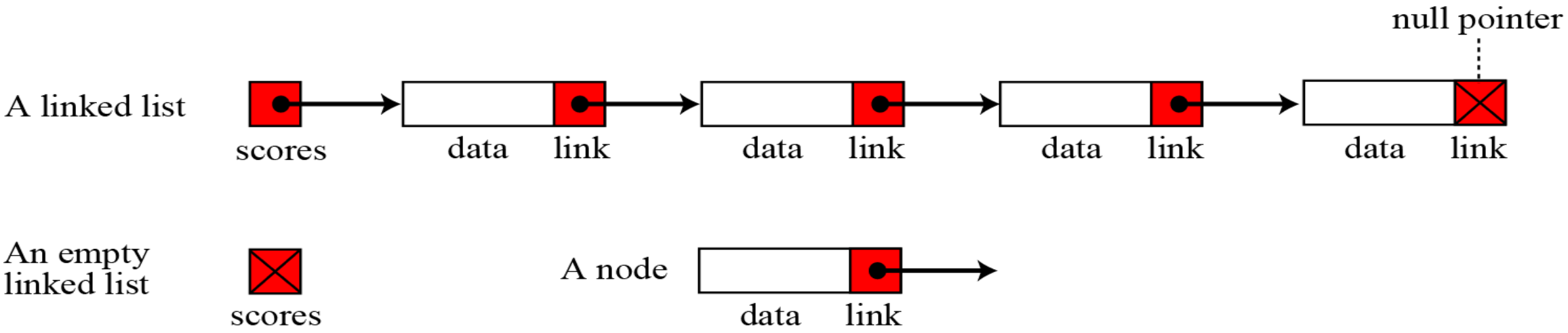
Module – 3
Linked Lists

Contents

- Introduction to Linked list
- Advantages and Disadvantages of Linked List
- Types of Linked List
- Single Linked List
- Applications of Linked List
- Circular Single Linked list
- Double Linked List

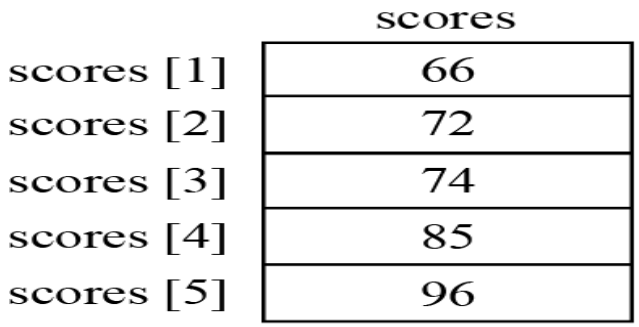
Introduction to Linked Lists

A linked list is a collection of data in which each element contains the location of the next element—that is, each element contains two parts: data and link.

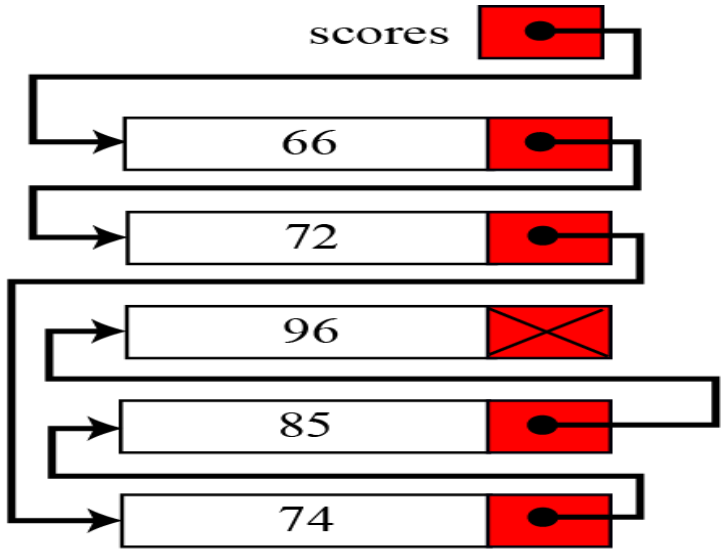


Arrays vs Linked Lists

- Both an array and a linked list are representations of a list of items in memory. The only difference is the way in which the items are linked together. The Figure below compares the two representations for a list of five integers.



a. Array representation



b. Linked list representation

Linked Lists – A Dynamic Data Structure

- A data structure that can shrink or grow during program execution.
- The size of a dynamic data structure is not necessarily known at compilation time, in most programming languages.
- Efficient insertion and deletion of elements.
- The data in a dynamic data structure can be stored in non-contiguous (arbitrary) locations.
- Linked list is an example of a dynamic data structure.

Advantages of Linked Lists

- Unused locations in array is often a wastage of space
- Linked lists offer an efficient use of memory
  - Create nodes when they are required
  - Delete nodes when they are not required anymore
  - We don’t have to know in advance how long the list should be

Applications of Linked Lists

- Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

  $P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

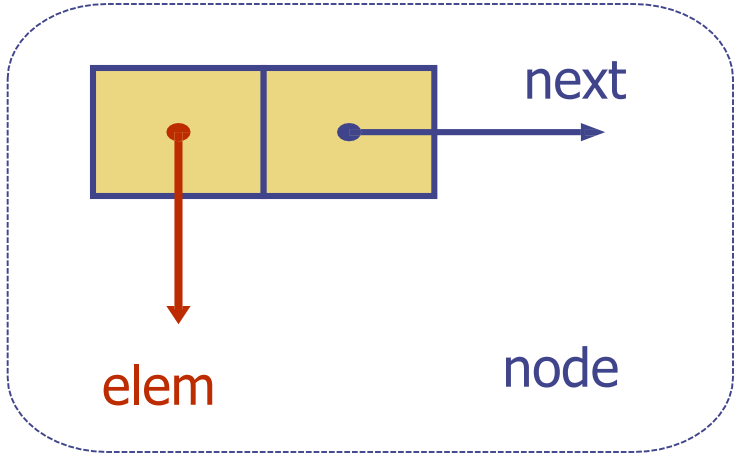
- Represent very large numbers and operations of the large number such as addition, multiplication and division.
- Linked lists are to implement stack, queue, trees and graphs.
- Implement the symbol table in compiler construction.

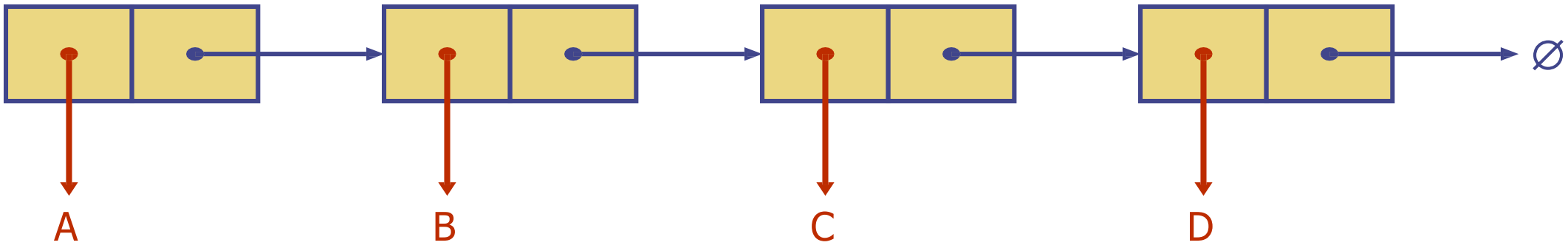
Types of Linked Lists

- There are four types of Linked lists:
  - Single linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - Circular single linked list
    - Pointer in the last node points back to the first node
  - Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - Circular double linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Single Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

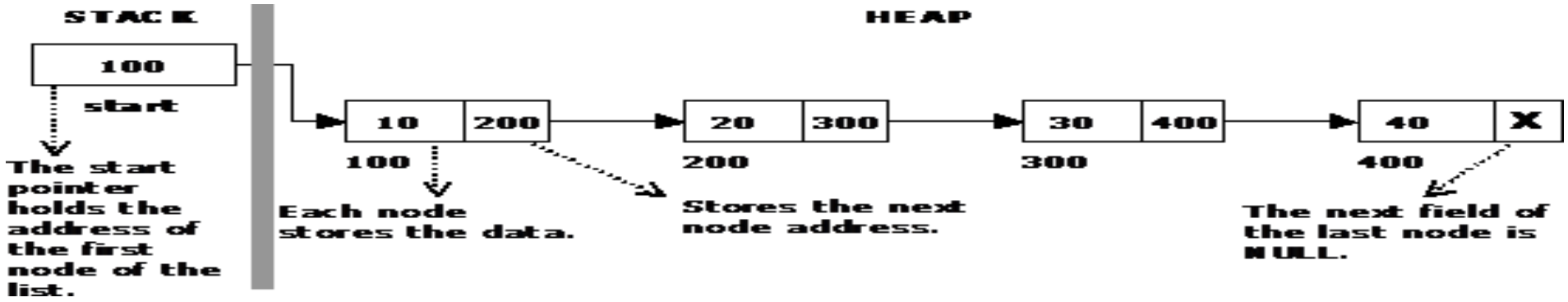




Single Linked List

- A linked list allocates space for each element separately in its own block of memory called a "node".
- Each node contains two fields; a "**data**" field to store whatever element, and a "**next**" field which is a pointer used to link to the next node.
- Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free().
- The front of the list is a pointer to the “start” node.

Single Linked List




```
struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```

node:

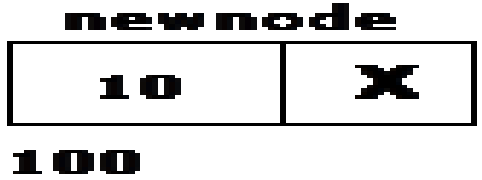
Empty list:

Operations on Single Linked List

- The basic operations of a single linked list are
  - Creation
  - Insertion
  - Deletion
  - Traversing

Creating a node for Single Linked List

Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.



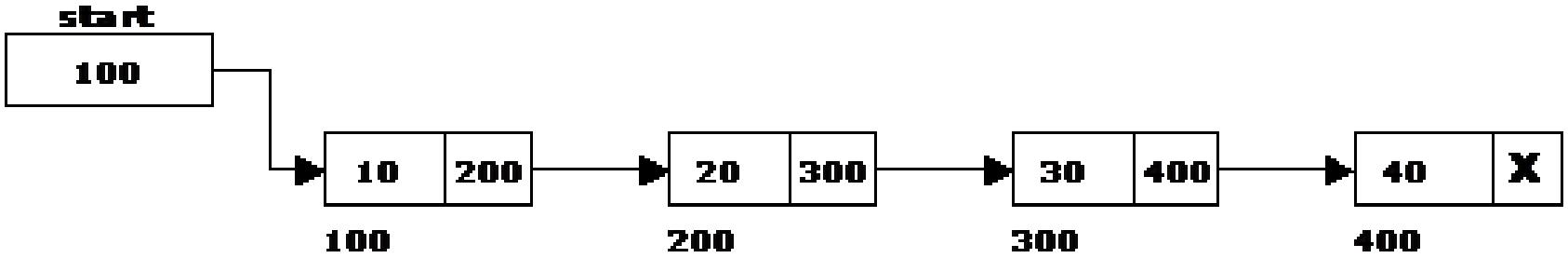
Single Linked List

```
class Node:
        def __init__(self, data, nextNode = None):
                self.data = data
                self.nextNode = nextNode
        def getData(self):
                return self.data
        def setData(self,val):
                self.data = val
        def getNextNode(self):
                return self.nextNode
        def setNextNode(self,val):
                self.nextNode = val
```

Single Linked List

```
class LinkedList:
        def __init__(self,head = None):
                self.head = head
                self.size = 0
        def getSize(self):
                return self.size
        def addNode(self,data):
                newNode = Node(data,self.head)
                self.head = newNode
                self.size+=1
                return True
        def printNode(self):
                curr = self.head
                while curr:
                        print(curr.data)
                        curr = curr.getNextNode()
```

Creating a Single Linked List with N nodes



Inserting a node into a Single Linked List

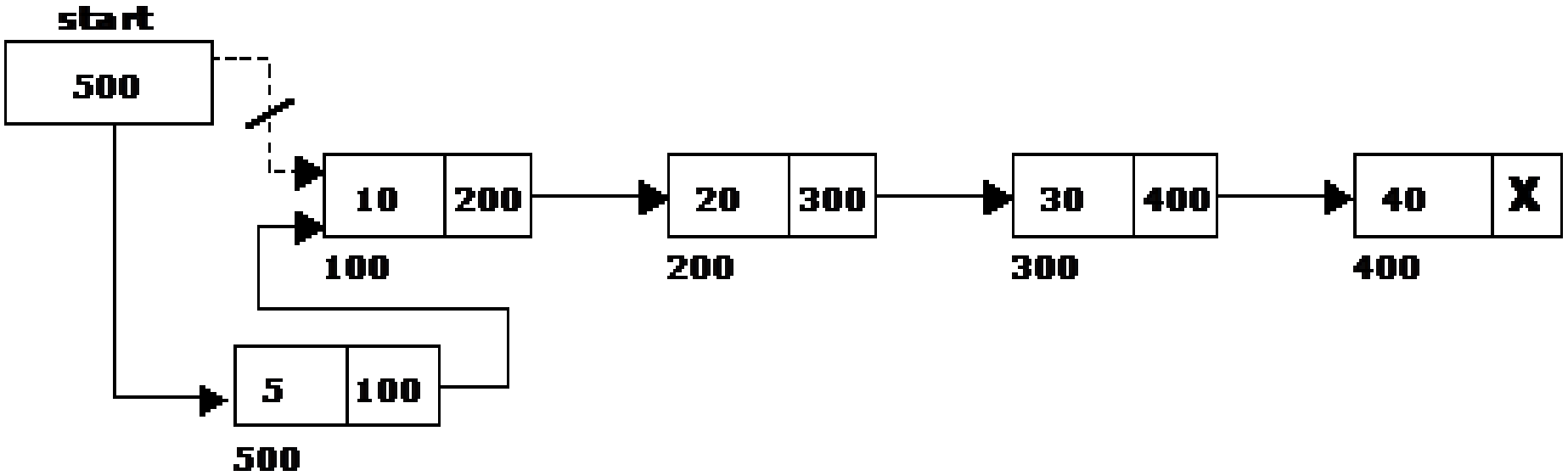
- Inserting a node into a single linked list can be done at
  - Inserting into an empty list.
  - Insertion at the beginning of the list.
  - Insertion at the end of the list.
  - Insertion in the middle of the list.

Inserting a node at the beginning

The following steps are to be followed to insert a new node at the beginning of the list:

```
#Function to insert a new node at the beginning
def push(self, new_data):
        # Allocate the Node & Put in the data
        new_node = Node(new_data)
        #Make next of new Node as head
        new_node.next = self.head
        # Move the head to point to new Node
        self.head = new_node
```

Inserting a node at the beginning



Inserting a node at the end

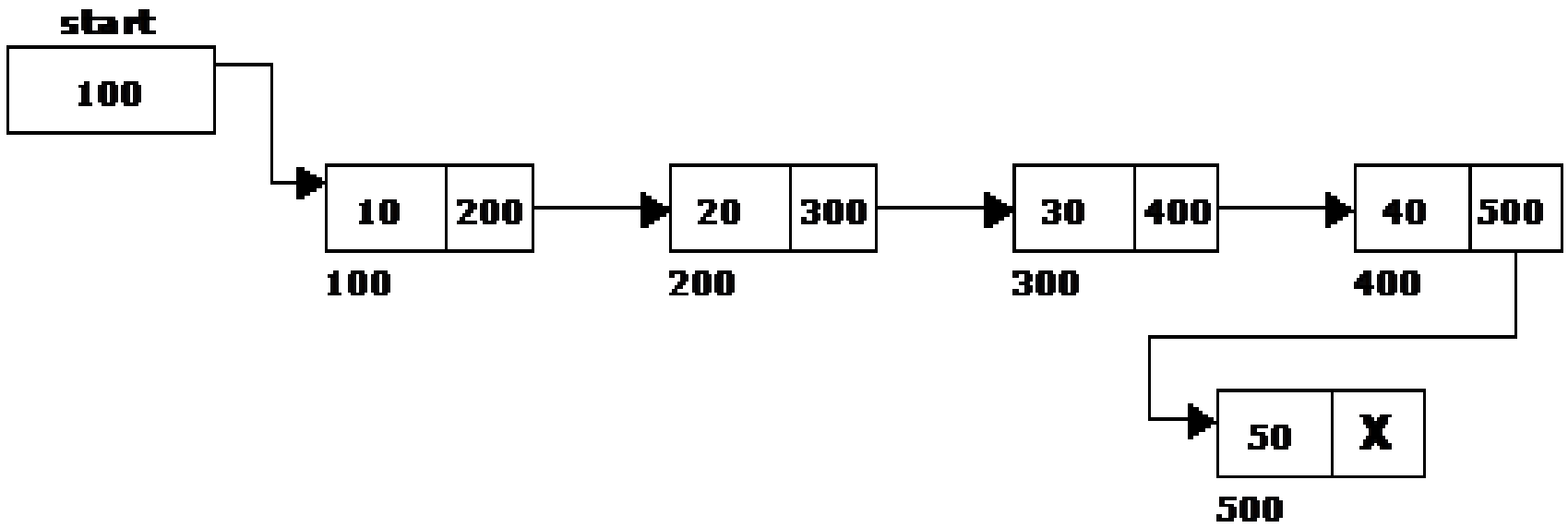
- The following steps are followed to insert a new node at the end of the list:

```
# This function is defined in Linked List class
# Appends a new node at the end.  This method is defined inside LinkedList class
shown above
def append(self, new_data):
        # Create a new node, Put in the data, Set next as None
        new_node = Node(new_data)
```

Inserting a node at the end

```
# If the Linked List is empty, then make the
   new node as head
if self.head is None:
        self.head = new_node
                return
#Else traverse till the last node
last = self.head
while last.next:
                last = last.next
# Change the next of last node
last.next =  new_node
```

Inserting a node at the end

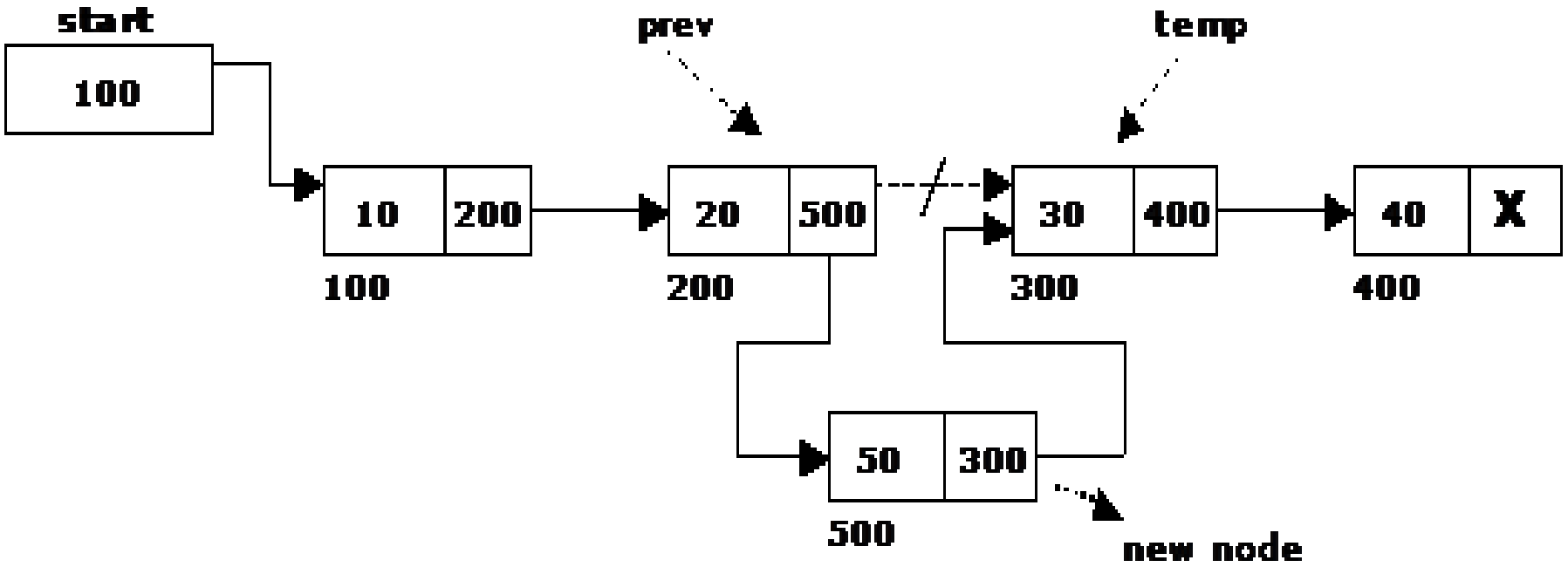


Inserting a node at intermediate position

- The following steps are followed, to insert a new node after the given previous node in the list:

```
def insertAfter(self, prev_node, new_data):
 #check if the given prev_node exists
 if prev_node is None:
                print("The given previous node must in Linked List.")
                return
 #Create new node & Put in the data
 new_node = Node(new_data)
 # Make next of new Node as next of prev_node
 new_node.next = prev_node.next
#Make next of prev_node as new_node
        prev_node.next = new_node
```

Inserting a node at intermediate position

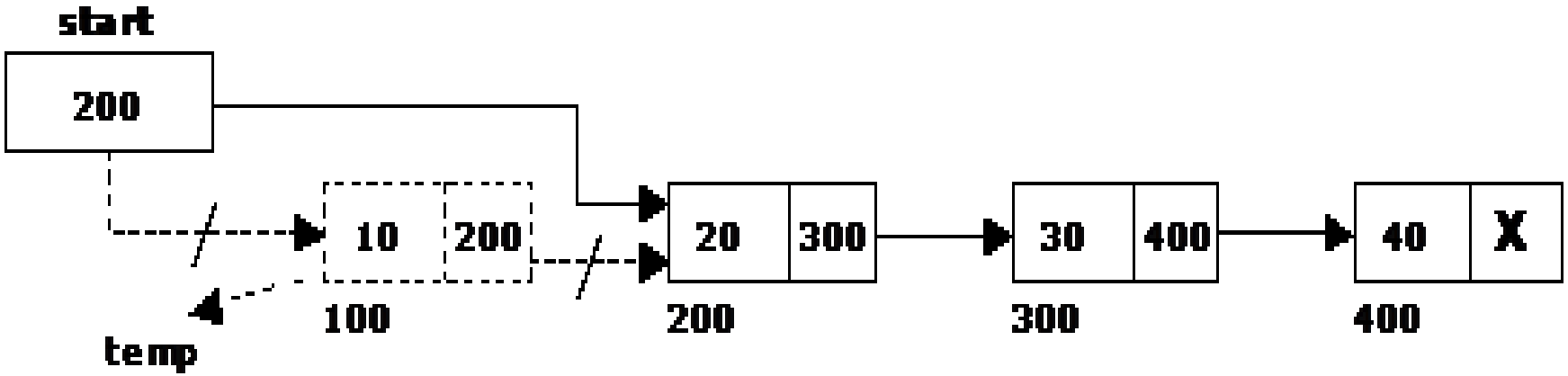


Deletion of a node

- Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.
  - Deleting a node at the beginning.
  - Deleting a node at the end.
  - Deleting a node at intermediate position.

Deleting a node at the beginning

- The following steps are followed, to delete a node at the beginning of the list:

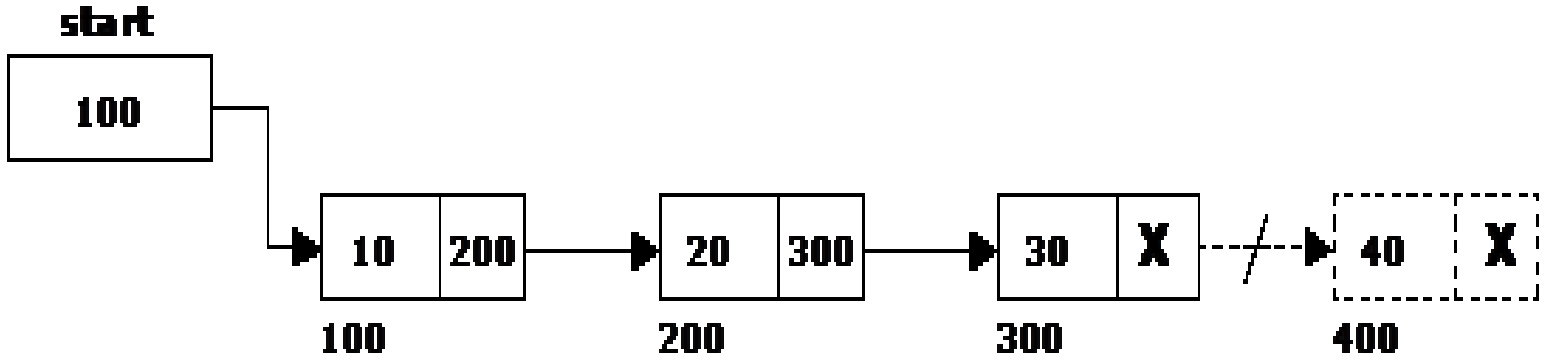


Deleting a node at the end

- The following steps are followed to delete a node at the end of the list:
  - If list is empty then display ‘Empty List’ message.
  - If the list is not empty, follow the steps given below:

```
temp = prev = start;  while(temp -> next != NULL)
{
prev = temp;
temp = temp -> next;
}
prev -> next = NULL;  free(temp);
```

Deleting a node at the end



Deleting a node at intermediate position

- The following steps are followed, to delete a node from an intermediate position in the list:

```
# Given a reference to the head of a list and a position, delete the node at a given position
def deleteNode(self, position):
        # If linked list is empty
        if self.head == None:
            return
        # Store head node
        temp = self.head
```

Deleting a node at intermediate position

```
# If head needs to be removed
                if position == 0:
                    self.head = temp.next
                        temp = None
                        return
# Find previous node of the node to be    deleted
        for i in range(position -1 ):
            temp = temp.next
            if temp is None:
                break
```

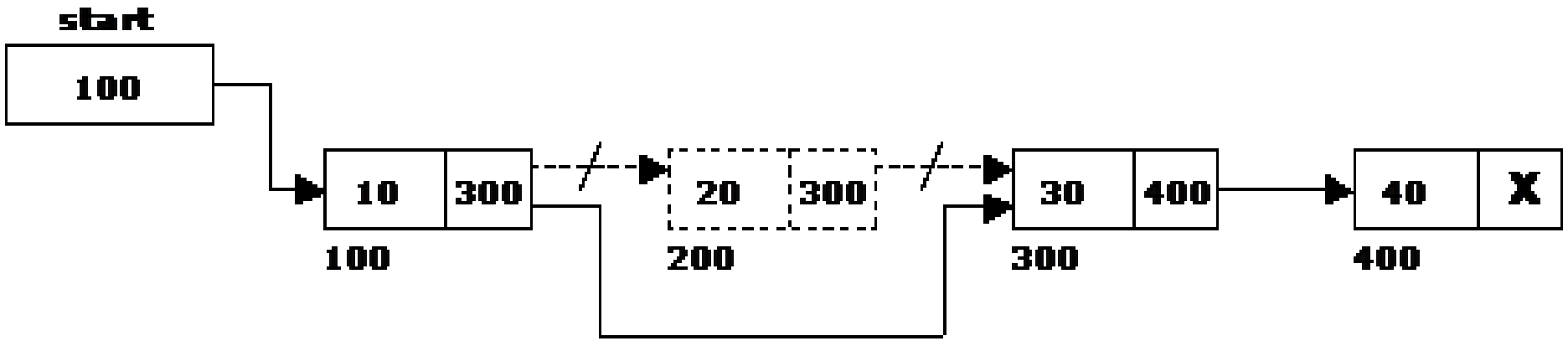
Deleting a node at intermediate position

```
# If position is more than number of nodes
    if temp is None:
        return
    if temp.next is None:
        return
   # Node temp.next is the node to be deleted
  store pointer to the next of node to be deleted
    next = temp.next.next
    # Unlink the node from linked list
    temp.next = None
temp.next=next
```

Deleting a node at intermediate position

```
# Find previous node of the node to be deleted
        for i in range(position -1 ):
            temp = temp.next
            if temp is None:
                break

        # If position is more than number of nodes
        if temp is None:
            return
        if temp.next is None:
            return
```

Deleting a node at intermediate position



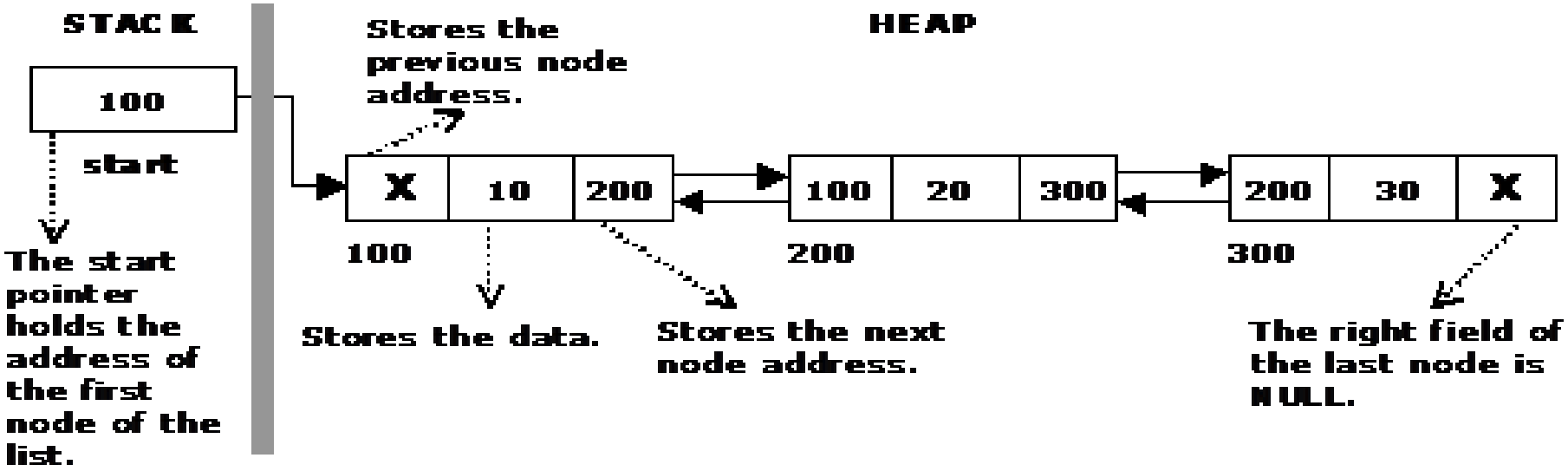
Traversal and displaying a list

- To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:
  - Assign the address of start pointer to a temp pointer.
  - Display the information from the data field of each node.

Double Linked list

- A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:
  - Left link.
  - Data.
  - Right link.
- The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Double Linked list



Operations on Double Linked list

- Creation
- Insertion
- Deletion
- Traversing

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

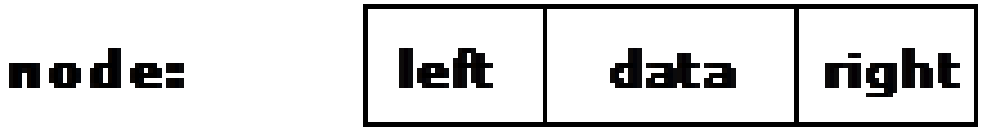
Creating a Double Linked list with n nodes

- The following steps are to be followed to create ‘n’ number of nodes:

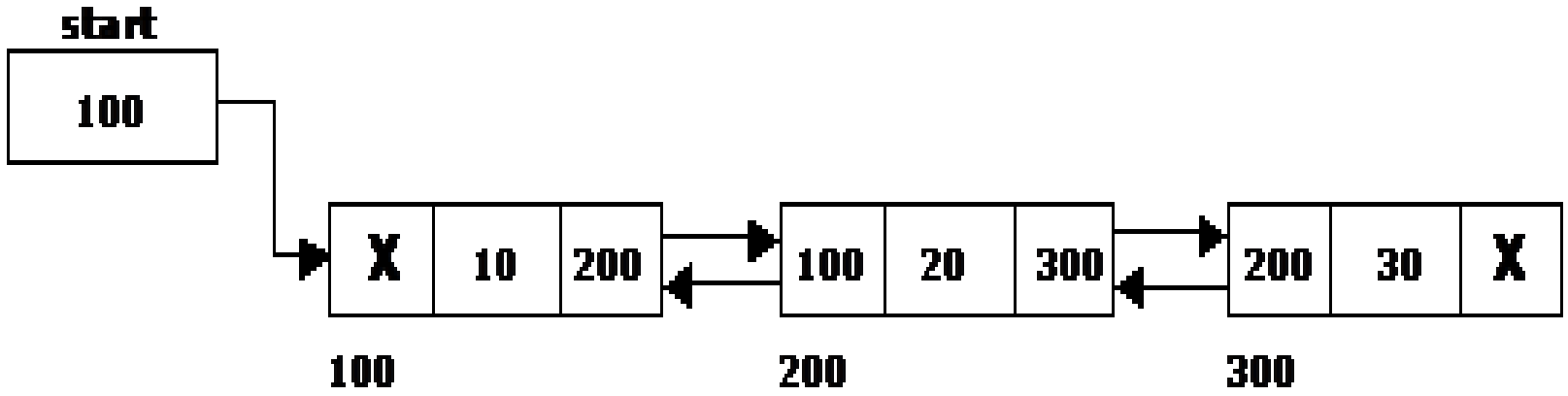
```
class Node(object):
        def __init__(self, data, prev, next):
            self.data = data
        self.prev = prev
        self.next = next

class DoubleList(object):
        head = None
tail = None
```

Structure of a Double Linked list

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```



Double Linked list with N nodes

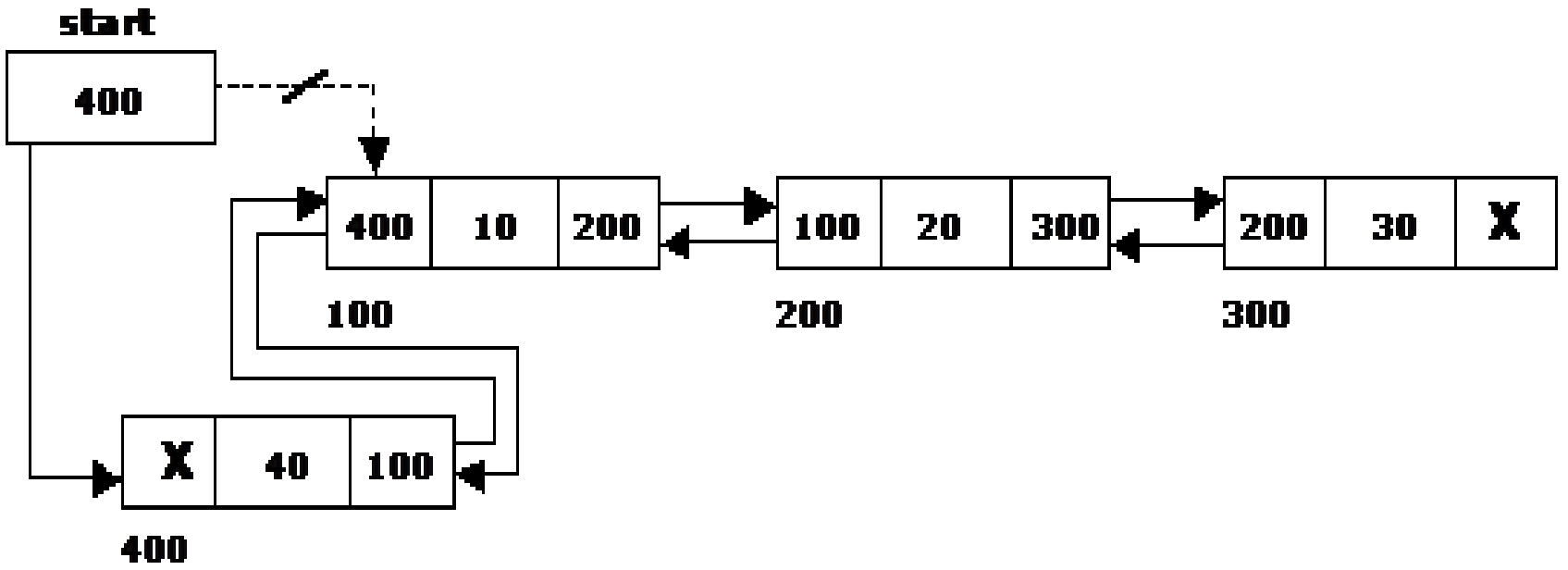


Insert a node at the beginning

- The following steps are to be followed to insert a new node at the beginning of the list:
- Get the new node using getnode(). newnode=getnode();
- If the list is empty then *start = newnode*.
- If the list is not empty, follow the steps given below:

newnode -> right = start; start -> left = newnode; start = newnode;

Insert a node at the beginning



Insert a node at the end

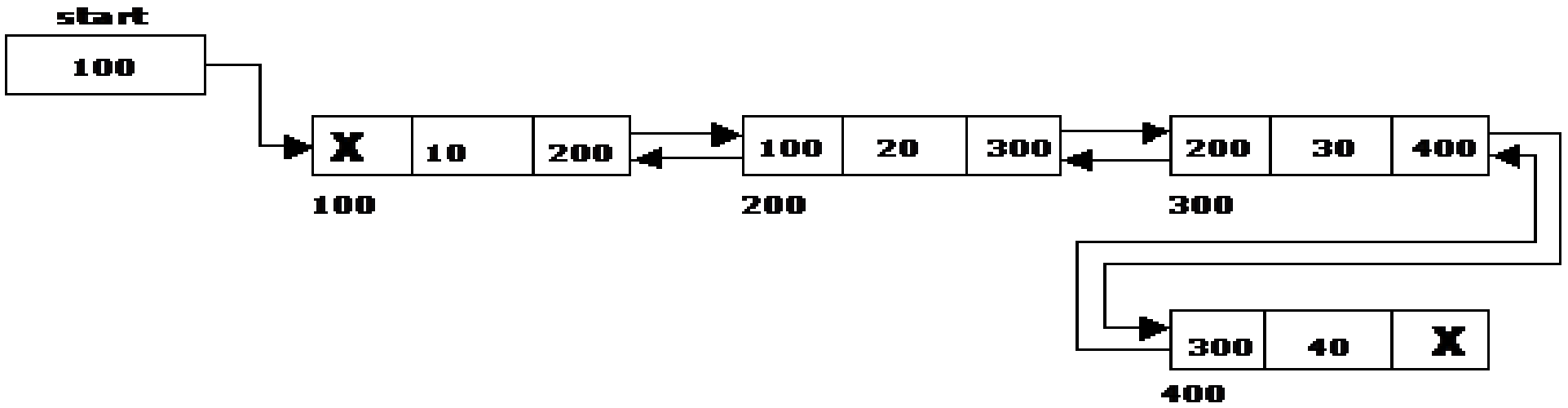
- The following steps are followed to insert a new node at the end of the list:
- Get the new node using getnode()

  newnode=getnode();
- If the list is empty then *start = newnode*.
- If the list is not empty follow the steps given below: temp = start;

  while(temp -> right != NULL) temp = temp -> right;

  temp -> right = newnode; newnode -> left = temp;

Insert a node at the end

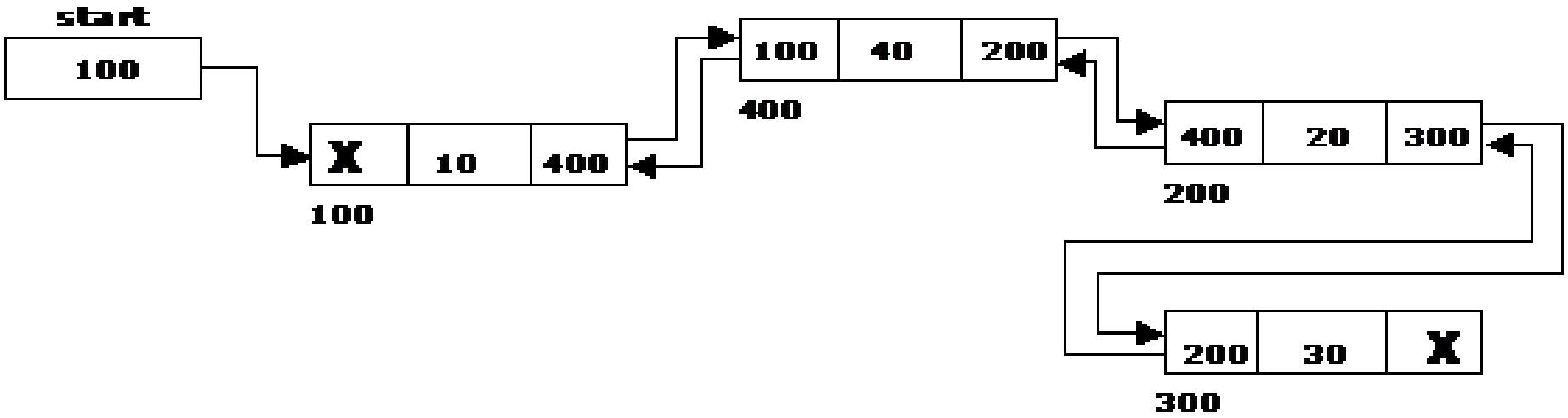


Insert a node at intermediate position

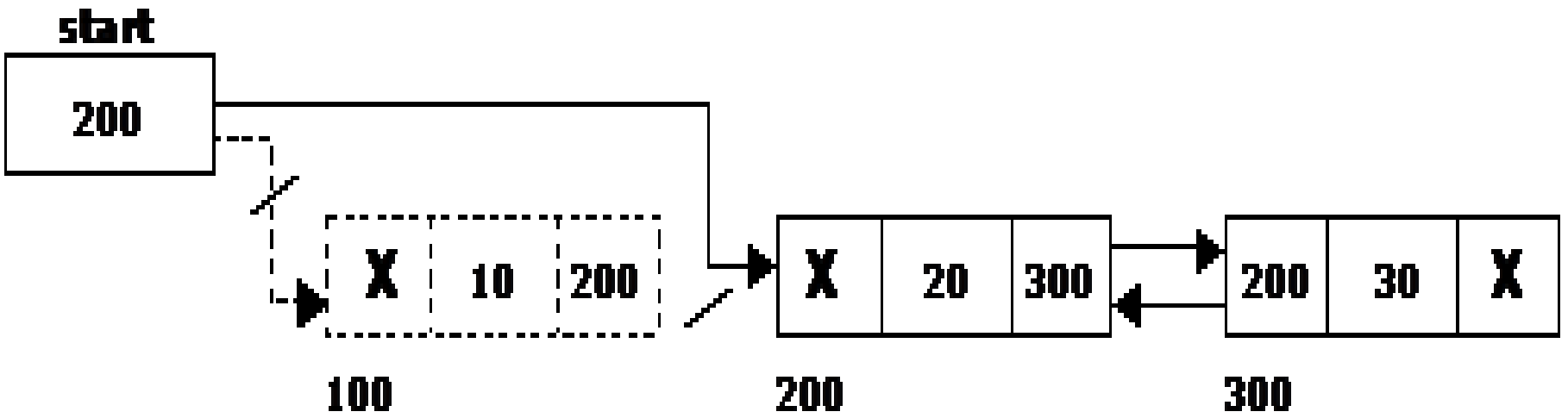
- The following steps are followed, to insert a new node in an intermediate position in the list:
- Get the new node using getnode(). newnode=getnode();
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below: newnode -> left = temp;

newnode -> right = temp -> right; temp -> right -> left = newnode; temp -> right = newnode;

Insert a node at intermediate position



Delete a node at beginning

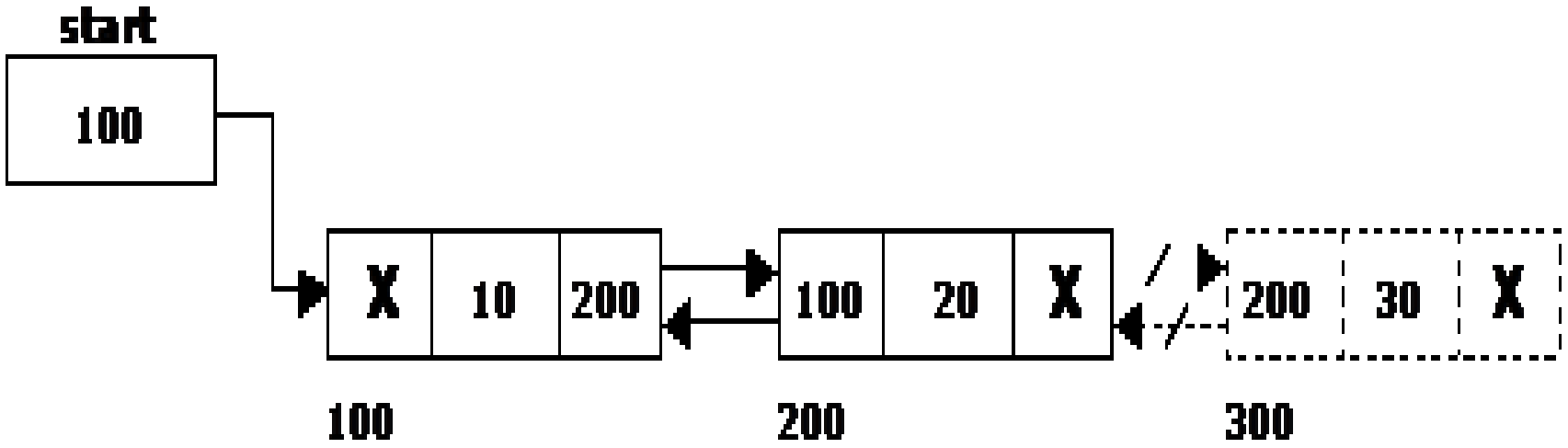


Delete a node at the end

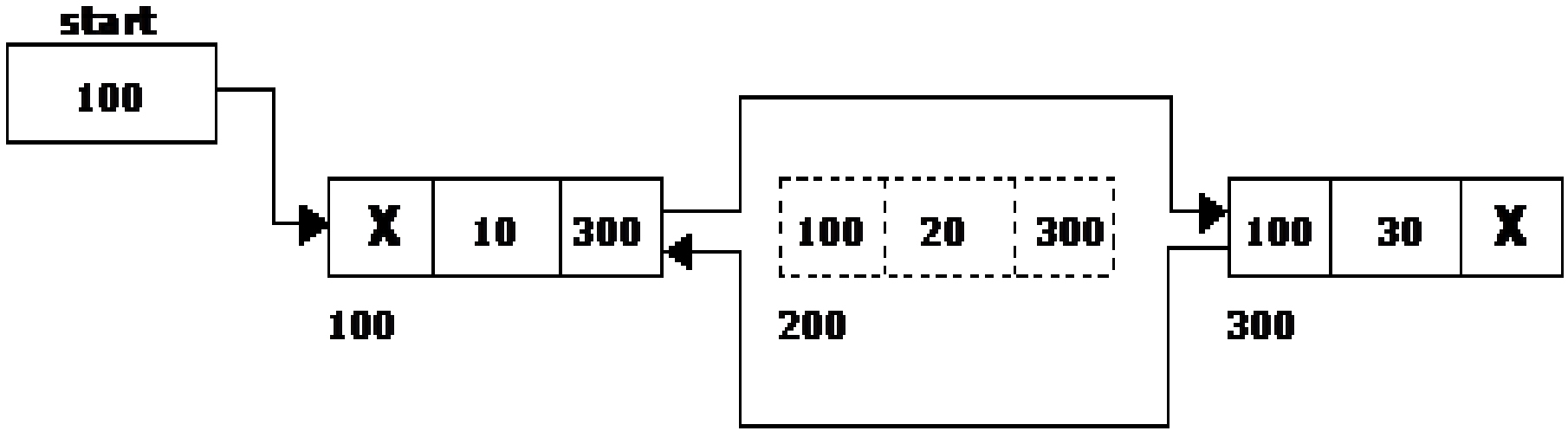
- The following steps are followed to delete a node at the end of the list:
  - If list is empty then display ‘Empty List’ message
  - If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
temp = temp -> right;
}
temp -> left -> right = NULL;  free(temp);
```

Delete a node at the end



Delete a node at intermediate position



Traversing a linked list from L to R

- The following steps are followed, to traverse a list from left to right:
- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below: temp = start;

```
while(temp != NULL)
{
print     temp -> data;  temp = temp -> right;
}
```

Traversing a linked list from R to L

- The following steps are followed, to traverse a list from right to left:
- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below: temp = start;

while(temp -> right != NULL) temp = temp -> right; while(temp != NULL)
{
print temp -> data; temp = temp -> left;
}

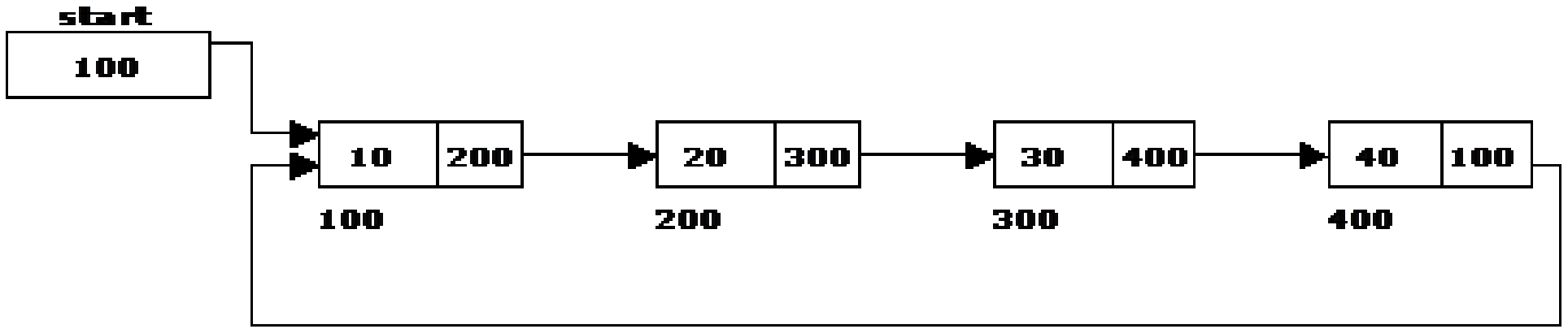
Advantages and Disadvantages of Double Linked list

- The **major disadvantage** of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.
- The **major advantage** of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

Circular Single Linked list

- It is just a single linked list in which the link field of the last node points back to the address of the first node.
- A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list.
- Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

Circular Single Linked list Operations



The basic operations in a circular single linked list are:

- Creation
- Insertion
- Deletion
- Traversing

Circular Single Linked list with n nodes

- The following steps are to be followed to create ‘n’ number of nodes:
- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start. start = newnode;
- If the list is not empty, follow the steps given below: temp = start;

```
while(temp -> next != NULL)  temp = temp -> next;
temp -> next = newnode;
```

- Repeat the above steps ‘n’ times. newnode -> next = start;

Inserting a node at the beginning

- The following steps are to be followed to insert a new node at the beginning of the circular list:
- Get the new node using getnode().

```
newnode = getnode();
```

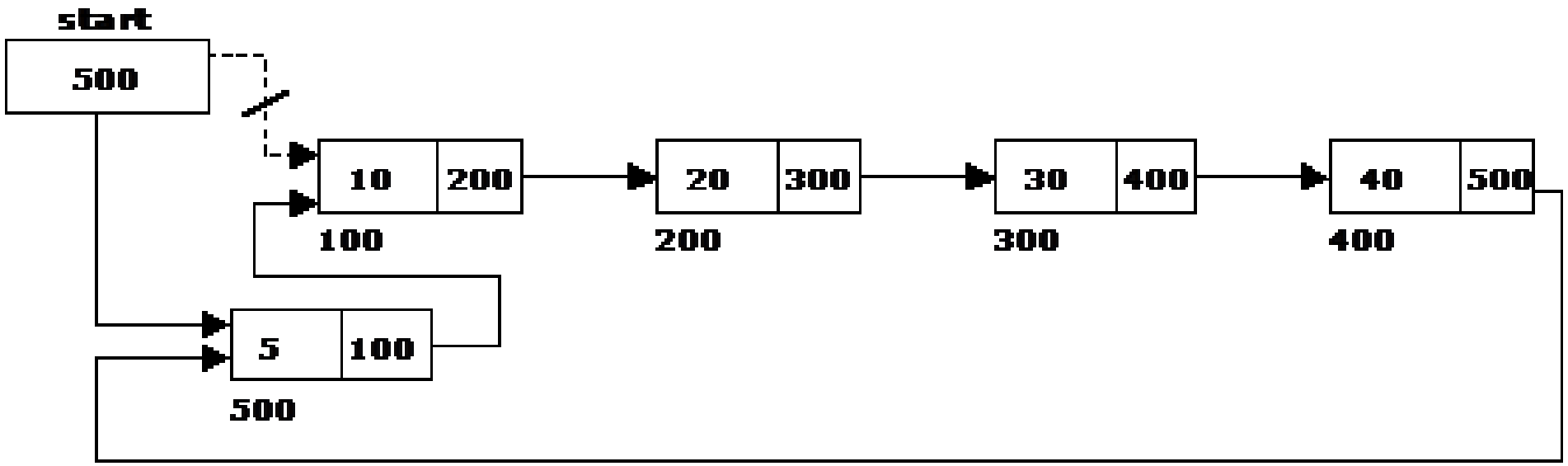
- If the list is empty, assign new node as start. start = newnode;

```
newnode -> next = start;
```

- If the list is not empty, follow the steps given below: last = start;

```
while(last -> next != start)  last = last -> next;
newnode -> next = start;  start = newnode;
last -> next = start;
```

Inserting a node at the beginning



Inserting a node at the end

- The following steps are followed to insert a new node at the end of the list:
- Get the new node using getnode(). newnode = getnode();
- If the list is empty, assign new node as start. start = newnode;

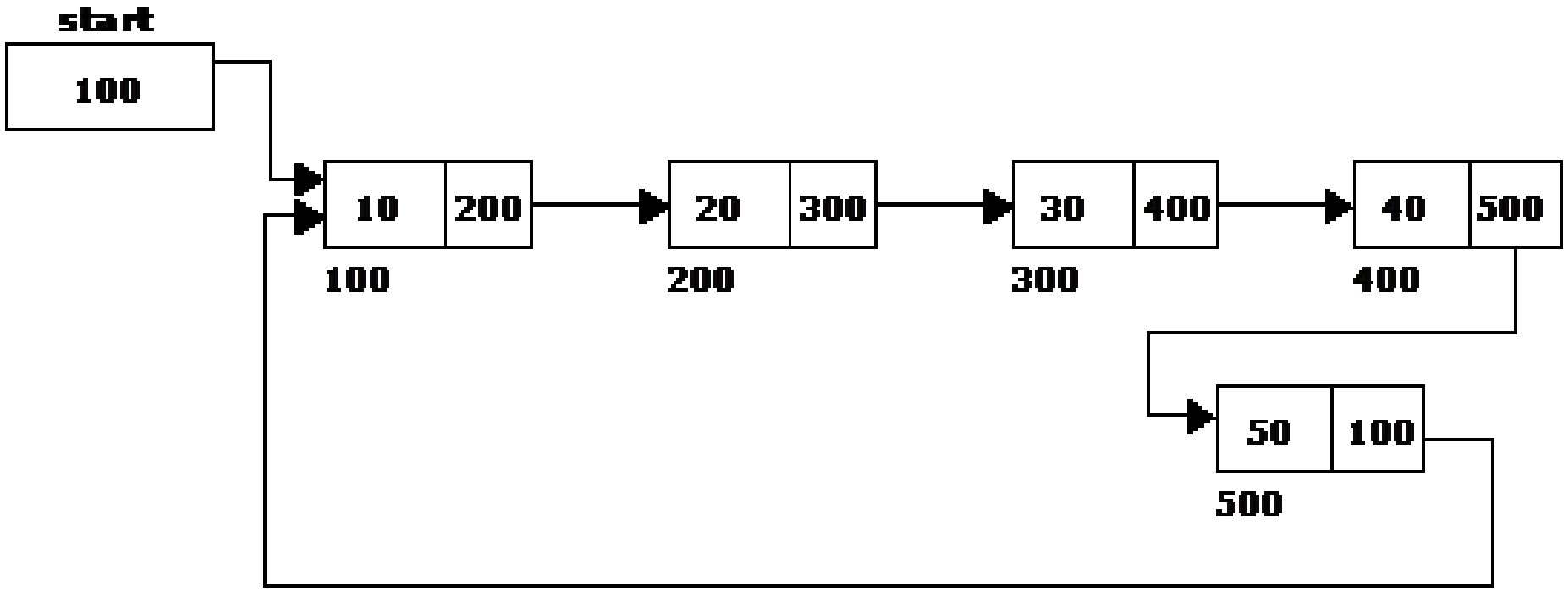
newnode -> next = start;

- If the list is not empty follow the steps given below: temp = start;

while(temp -> next != start) temp = temp -> next;

temp -> next = newnode; newnode -> next = start;

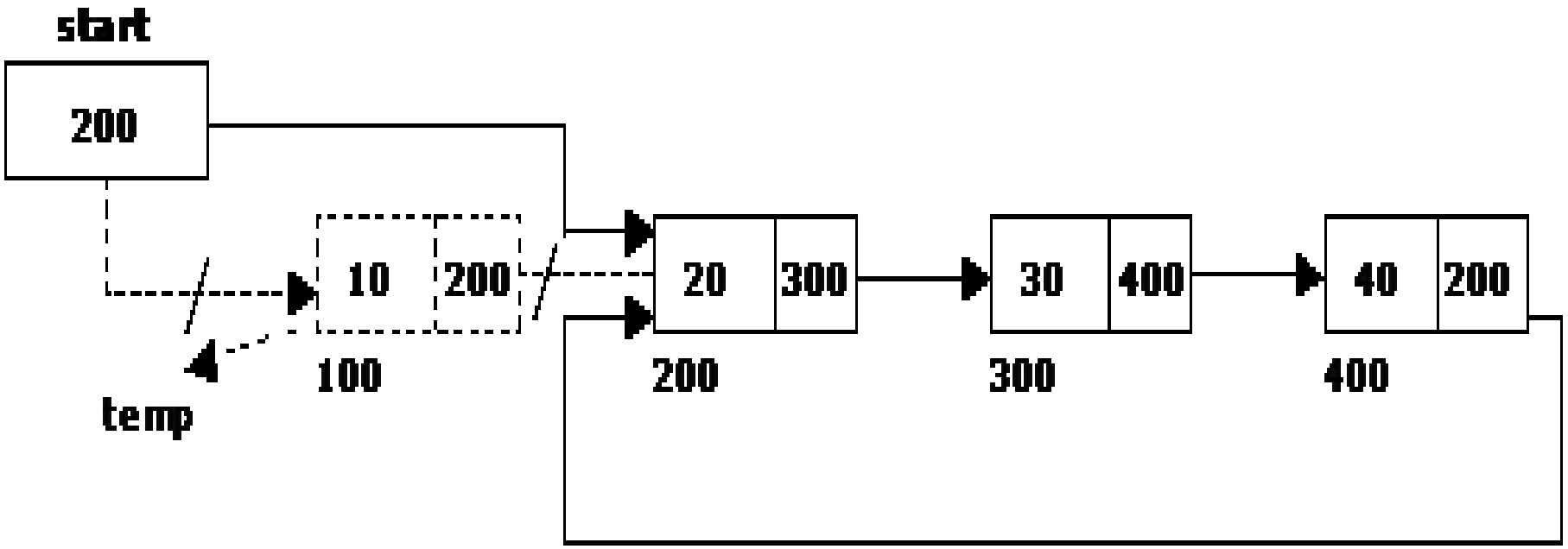
Inserting a node at the end



Deleting a node at the beginning

- The following steps are followed, to delete a node at the beginning of the list:
- If the list is empty, display a message ‘Empty List’.
- If the list is not empty, follow the steps given below: last = temp = start;
  while(last -> next != start) last = last -> next;
  start = start -> next; last -> next = start;
- After deleting the node, if the list is empty then *start = NULL.*

Deleting a node at the beginning



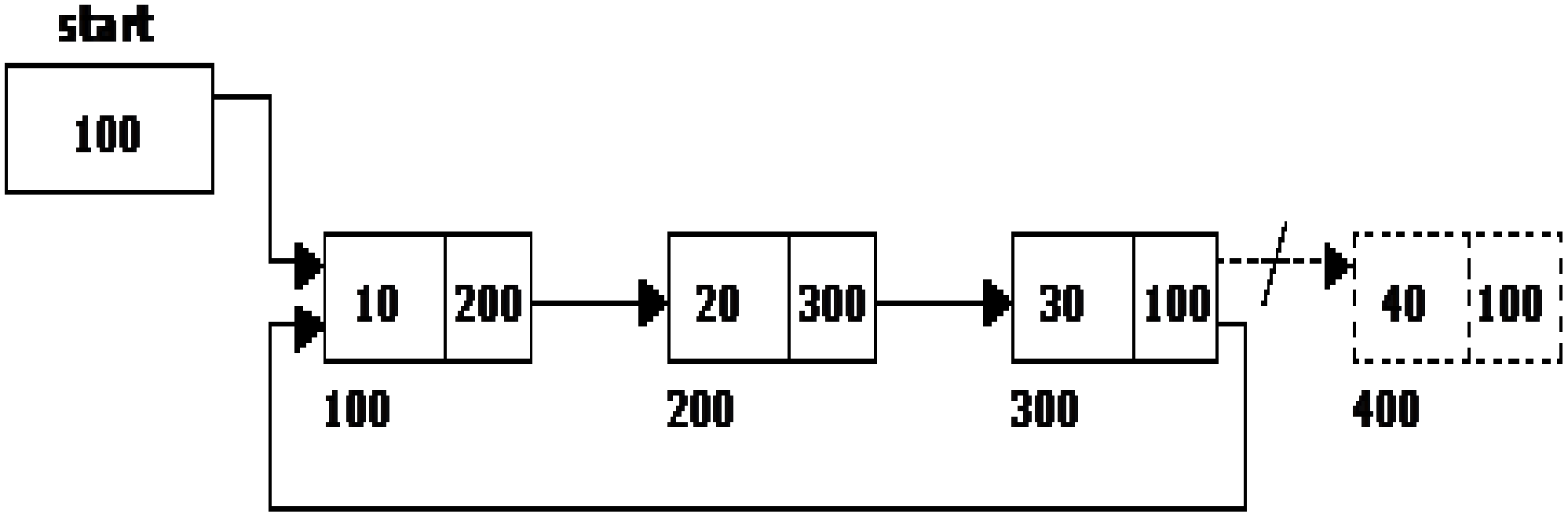
Deleting a node at the end

- The following steps are followed to delete a node at the end of the list:
- If the list is empty, display a message ‘Empty List’.
- If the list is not empty, follow the steps given below: temp = start;

```
prev = start;
while(temp -> next != start)
{
prev = temp;
temp = temp -> next;
}
prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL*.

Deleting a node at the end



Traversing a circular single linked list from left to right

- The following steps are followed, to traverse a list from left to right:
- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below: temp = start;

```
do
{
printf("%d ", temp -> data);  temp = temp -> next;
} while(temp != start);
```

Advantages of Circular Lists

- The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node).
- Also, some applications lead naturally to circular list representations.
- For example, a computer network might best be modeled using a circular list.

Applications of Linked Lists: Representing Polynomials

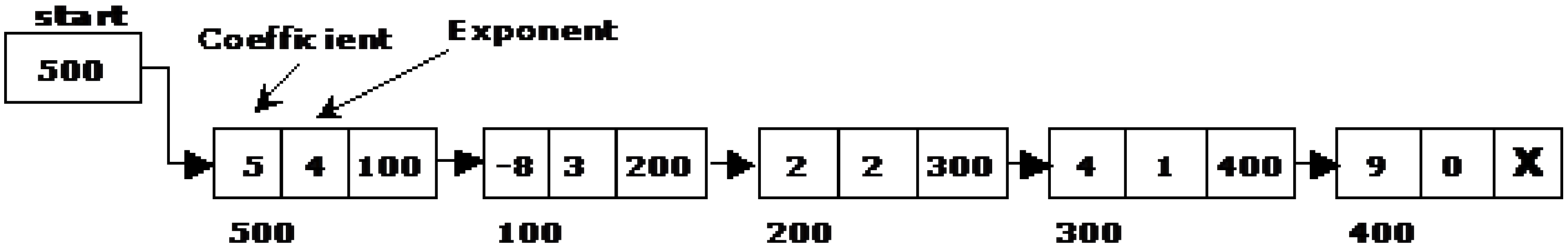
A polynomial is of the form: $\sum_{i=0}^{n} c_i x^i$

Where, c_i is the coefficient of the i^{th} term and n is the degree of the polynomial Some examples are:

$5x^2 + 3x + 1$

$5x^4 – 8x^3 + 2x^2 + 4x^1 + 9x^0$

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 – 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrated.



Addition of Polynomials

- To add two polynomials, if we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.
- To add two polynomials follow the following steps:
  - Read two polynomials.
  - Add them.
  - Display the resultant polynomial.

Module – 3
Linked Lists

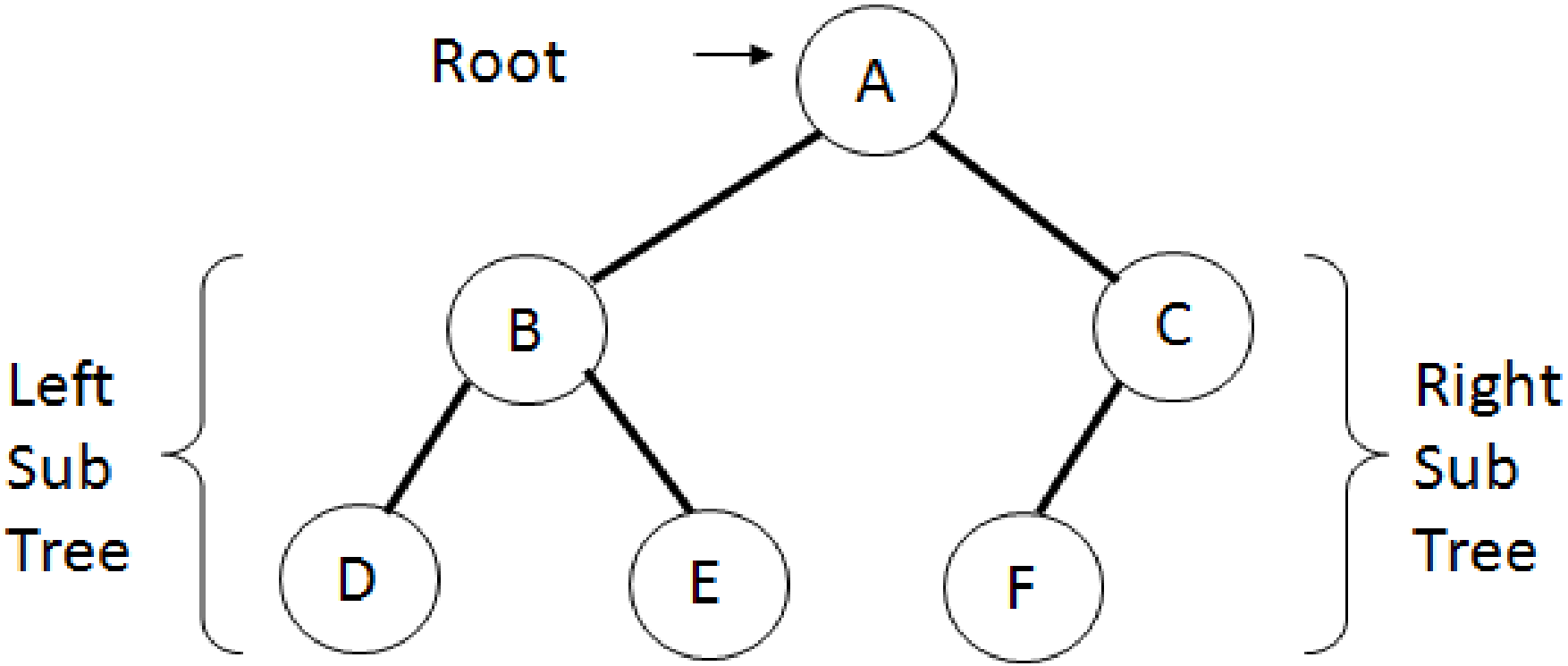
CONTENTS

- Basic Tree Concepts, Binary Trees
- Representation of Binary Trees
- Operations on a Binary Tree
- Binary Tree Traversals
- Threaded Binary Trees
- Basic Graph Concepts
- Graph Traversal Techniques: DFS and BFS

Tree – a Hierarchical Data Structure

- Trees are non linear data structure that can be represented in a hierarchical manner.
  - A tree contains a finite non-empty set of elements.
  - Any two nodes in the tree are connected with a relationship of parent-child.
  - Every individual elements in a tree can have any number of sub trees.

An Example of a Tree



Tree – Basic Terminology

- **Root :** The basic node of all nodes in the tree. All operations on the tree are performed with passing root node to the functions.
- **Child :** a successor node connected to a node is called child. A node in binary tree may have at most two children.
- **Parent :** a node is said to be parent node to all its child nodes.
- **Leaf :** a node that has no child nodes.
- **Siblings :** Two nodes are siblings if they are children to the same parent node.

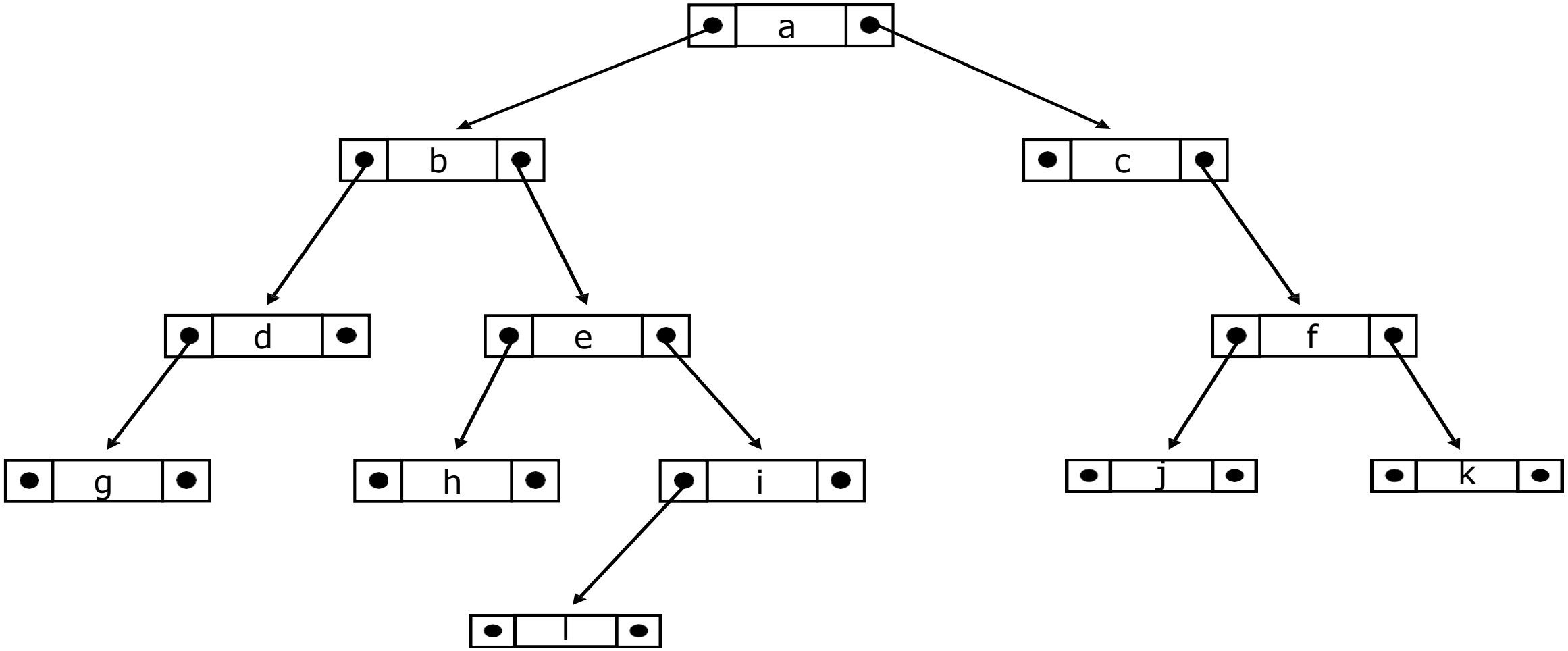
Tree – Basic Terminology Contd…

- **Ancestor :** a node which is parent of parent node (A is ancestor node to D,E and F).
- **Descendent :** a node which is child of child node (D, E and F are descendent nodes of node A)
- **Level** : The distance of a node from the root node, The root is at level – 0,(B and C are at Level 1 and D, E, F have Level 2 (highest level of tree is called **height** of tree)
- **Degree :** The number of nodes connected to a particular parent node.

Binary Tree

- A binary tree is a hierarchy of nodes, where every parent node has at most two child nodes. There is a unique node, called the root, that does not have a parent.
- A binary tree can be defined recursively as
- **Root node**
- **Left subtree:** left child and all its descendants
- **Right subtree:** right child and all its descendants

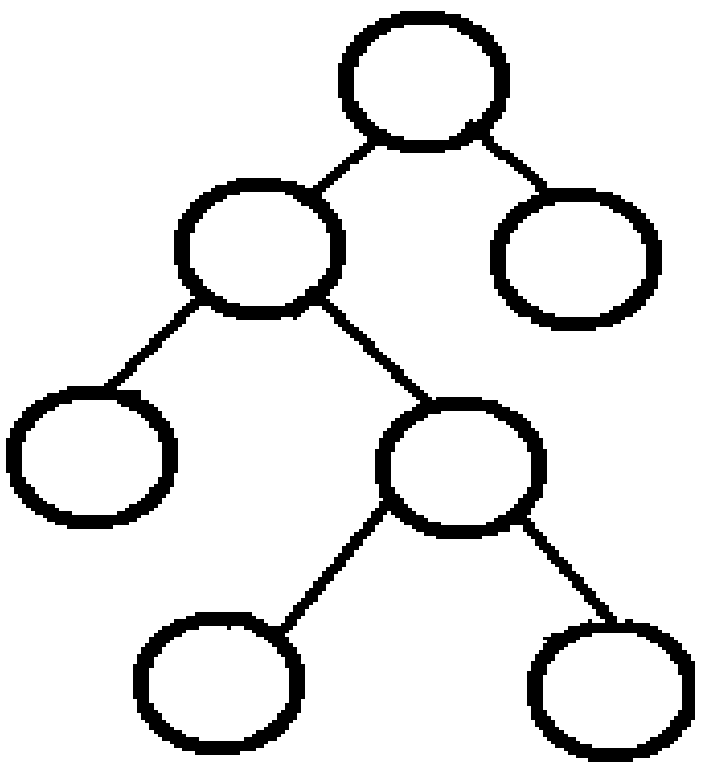
Binary Tree



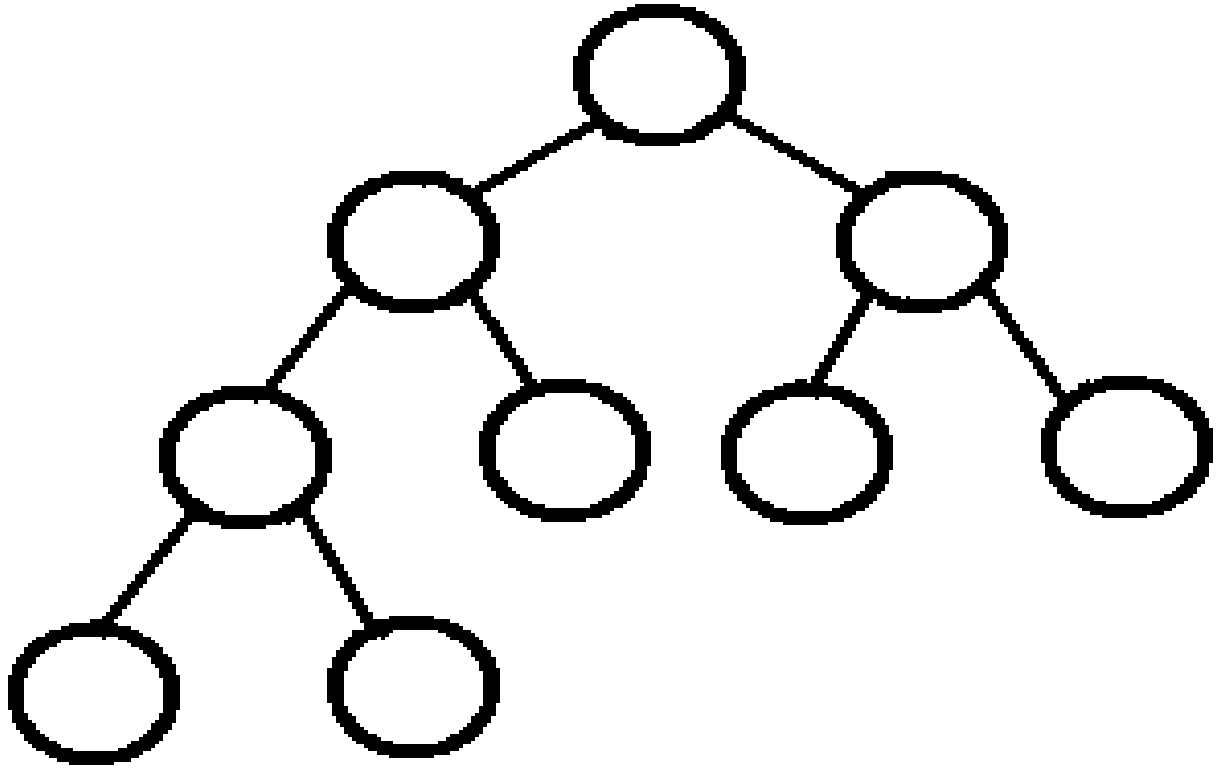
Full and Complete Binary Trees

- A **full tree is a binary tree in which**
  - Number of nodes at level *l is 2l–1*
  - Total nodes in a full tree of height *n is*
- A **complete tree of height *n is a binary tree***
  - Number of nodes at level 1 *l* *n–1 is 2l–1*
  - Leaf nodes at level *n occupy the* leftmost positions in the tree

Full and Complete Binary Trees



full tree



complete tree

Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree.
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Standard traversal orderings:
  - **preorder**
  - **inorder**
  - **postorder**
  - **level-order**

Preoder, Inorder, Postorder

- In Preorder, the root is visited before (pre)

the subtrees traversals.

- In Inorder, the root is visited in-between left

and right subtree traversal.

- In Preorder, the root is visited after (pre)

the subtrees traversals.

Preorder Traversal:
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Inorder Traversal:
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

Postorder Traversal:
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

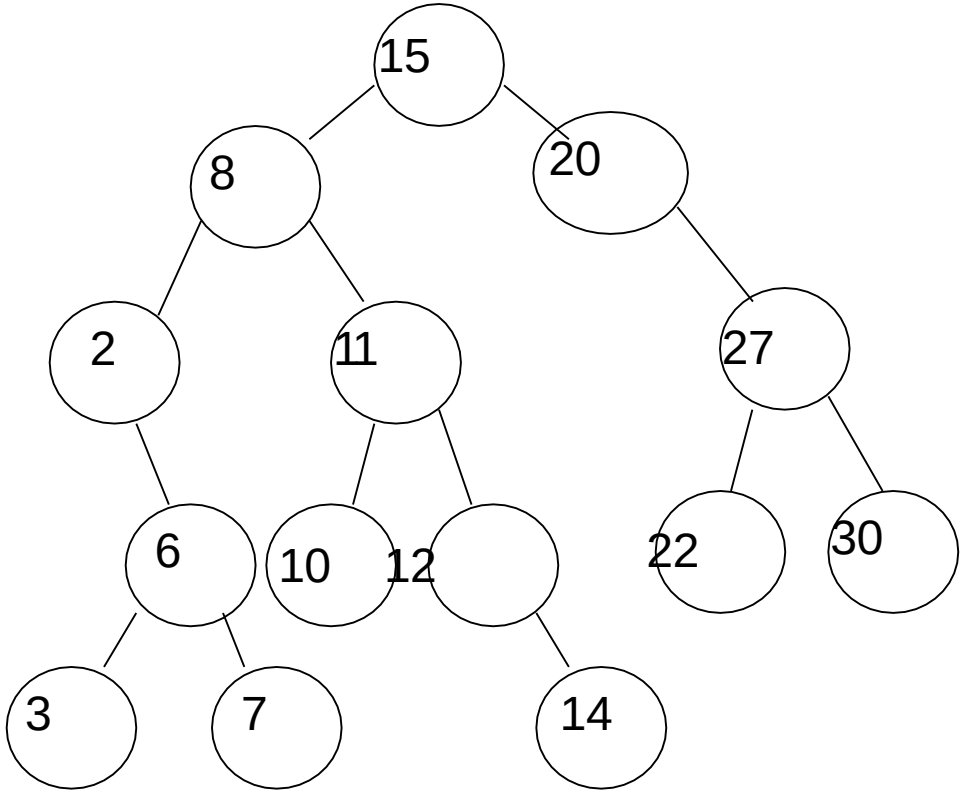
Example of Tree Traversal

- Assume: visiting a node is printing its data

•Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30

•Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30

•Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15



Traversal Techniques

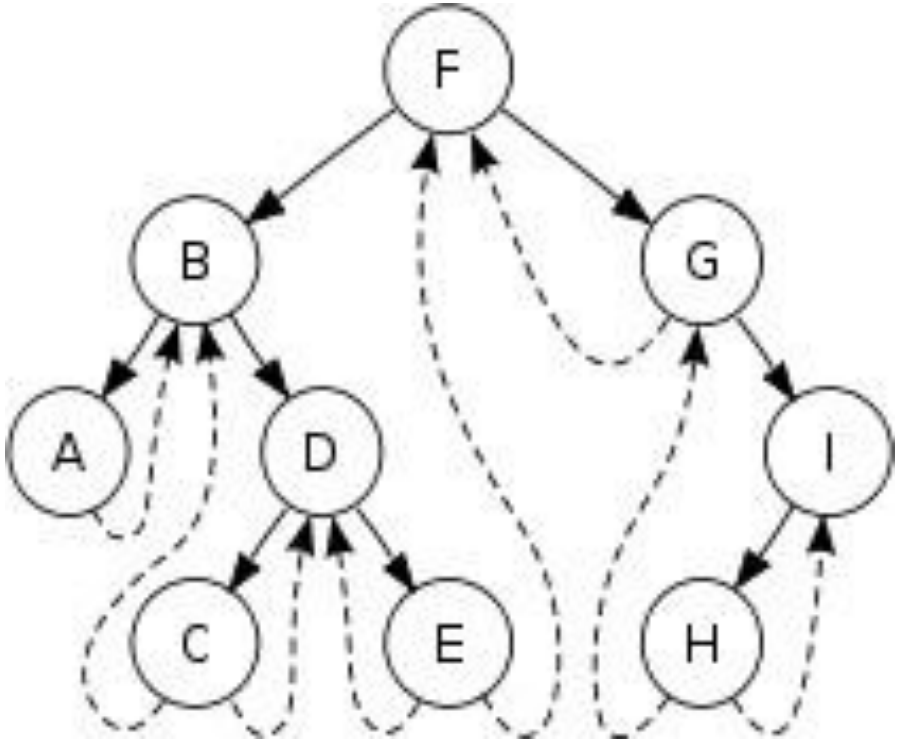
```
void preorder(tree *tree) {
if (tree->isEmpty( ))      return;  visit(tree->getRoot( ));  preOrder(tree->getLeftSubtree());
preOrder(tree->getRightSubtree());
}
```

```
void inOrder(Tree *tree){
if (tree->isEmpty( ))      return;  inOrder(tree->getLeftSubtree( ));  visit(tree->getRoot( ));
inOrder(tree->getRightSubtree( ));
}
```

```
void postOrder(Tree *tree){
if (tree->isEmpty( ))      return;  postOrder(tree->getLeftSubtree( ));  postOrder(tree->getRightSubtree(
));  visit(tree->getRoot( ));
}
```

Threaded Binary Tree

- A **threaded binary tree** defined as:
- "A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node



Graph Basics

- Graphs are collections of nodes connected by edges – G = (V,E) where V is a set of nodes and E a set of edges.
- Graphs are useful in a number of applications including
  - Shortest path problems
  - Maximum flow problems
- Graphs unlike trees are more general for they can have connected components.

Graph Types

- **Directed Graphs**: A directed graph edges allow travel in one direction.

- **Undirected Graphs**: An undirected edges allow graph travel in either direction.

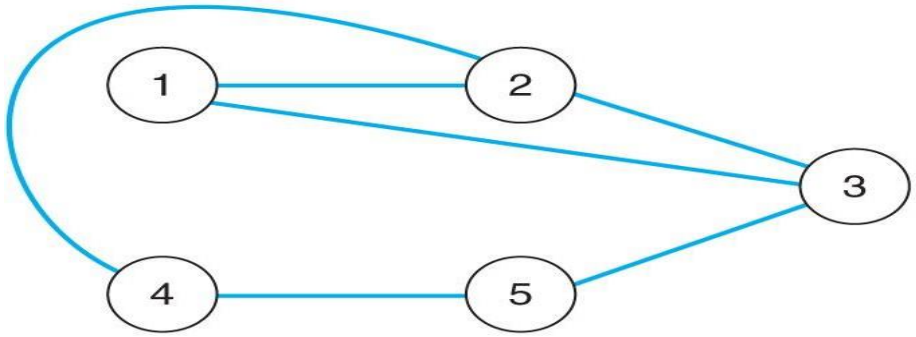


FIGURE 8.1A
The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

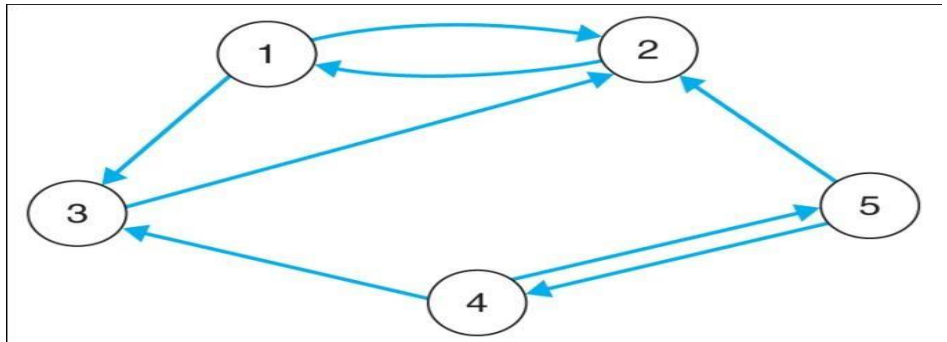


FIGURE 8.1B
The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

Graph Terminology

- A graph is an ordered pair G=(V,E) with a set of vertices or nodes and the edges that connect them.
- A subgraph of a graph has a subset of the vertices and edges.
- The edges indicate how we can move through the graph.
- A path is a subset of E that is a series of edges between two nodes.
- A graph is connected if there is at least one path between every pair of nodes.

Graph Terminology

- The length of a path in a graph is the number of edges in the path.
- A complete graph is one that has an edge between every pair of nodes.
- A weighted graph is one where each edge has a cost for traveling between the nodes.
- A cycle is a path that begins and ends at the same node.
- An acyclic graph is one that has no cycles.
- An acyclic, connected graph is also called an unrooted tree

Data Structures for Graphs: Adjacency Matrix

- For an undirected graph, the matrix will be symmetric along the diagonal.
- For a weighted graph, the adjacency matrix would have the weight for edges in the graph, zeros along the diagonal, and infinity (∞) every place else.

Adjacency Matrix

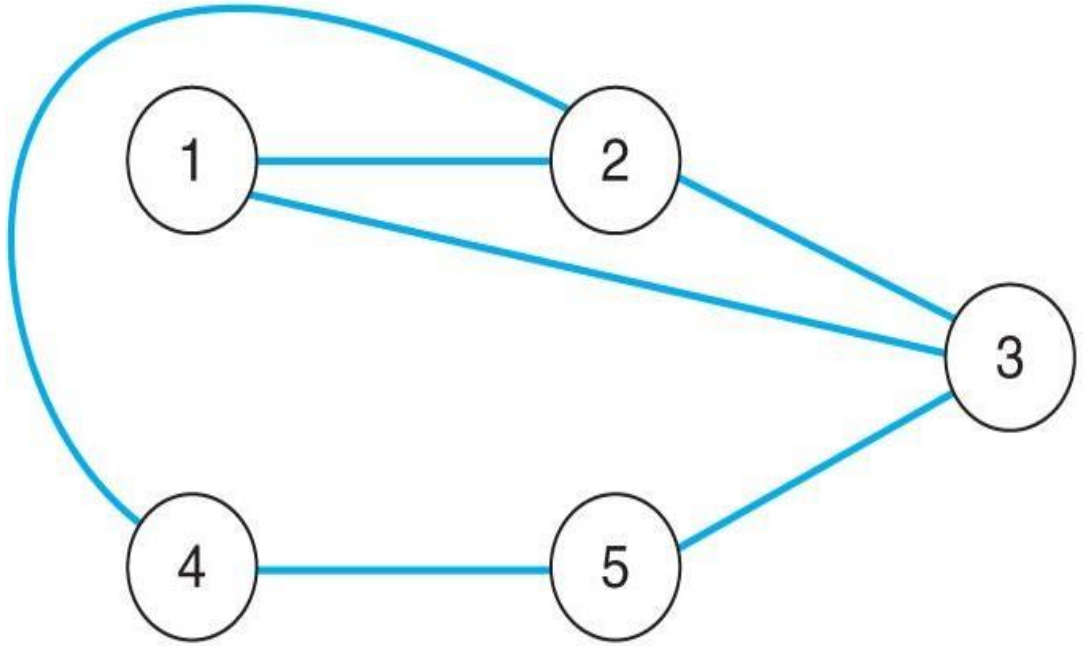


FIGURE 8.1A

The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	1
5	0	0	1	1	0

FIGURE 8.2A

The adjacency matrix for the graph in Fig. 8.1(a)

Adjacency Matrix

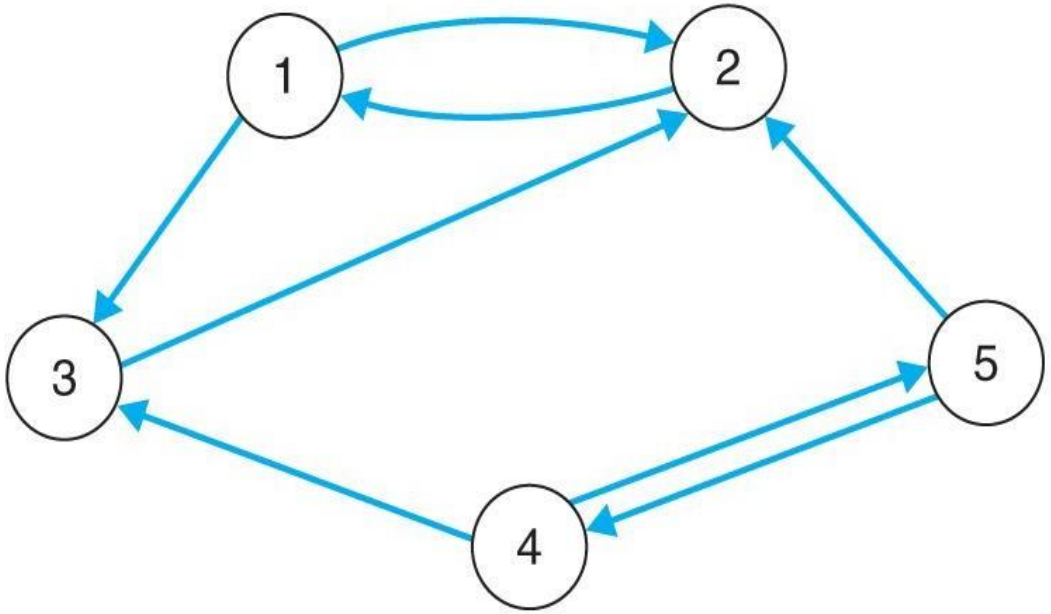


FIGURE 8.1B

The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	1	0	1	0

FIGURE 8.2B

The adjacency matrix for the digraph in Fig. 8.1(b)

Adjacency List

- A list of pointers, one for each node of the graph.
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph.
- For a weighted graph, this list would also include the weight for each edge.

Adjacency List

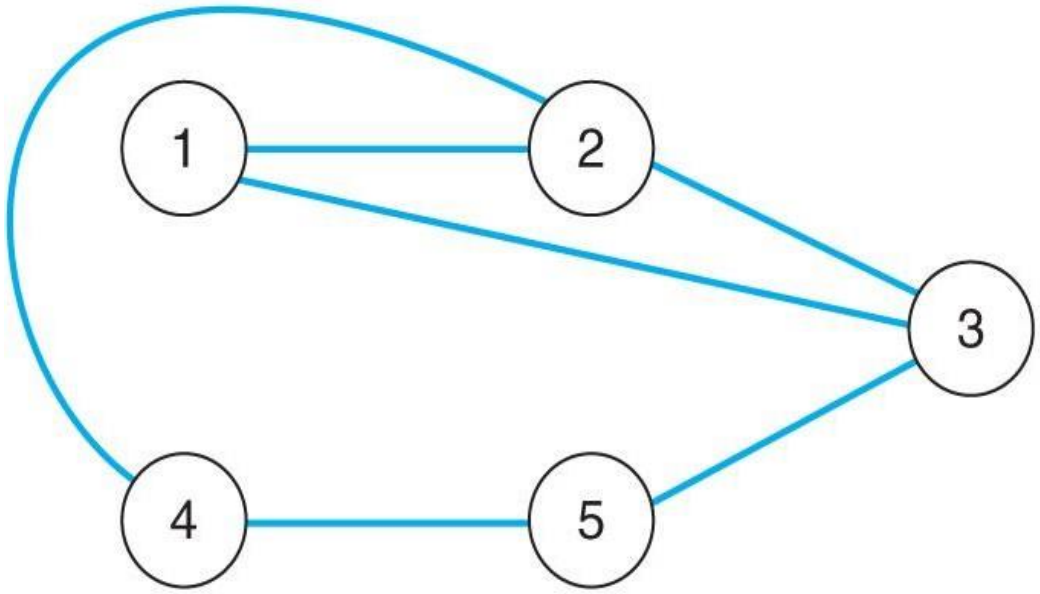


FIGURE 8.1A

The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

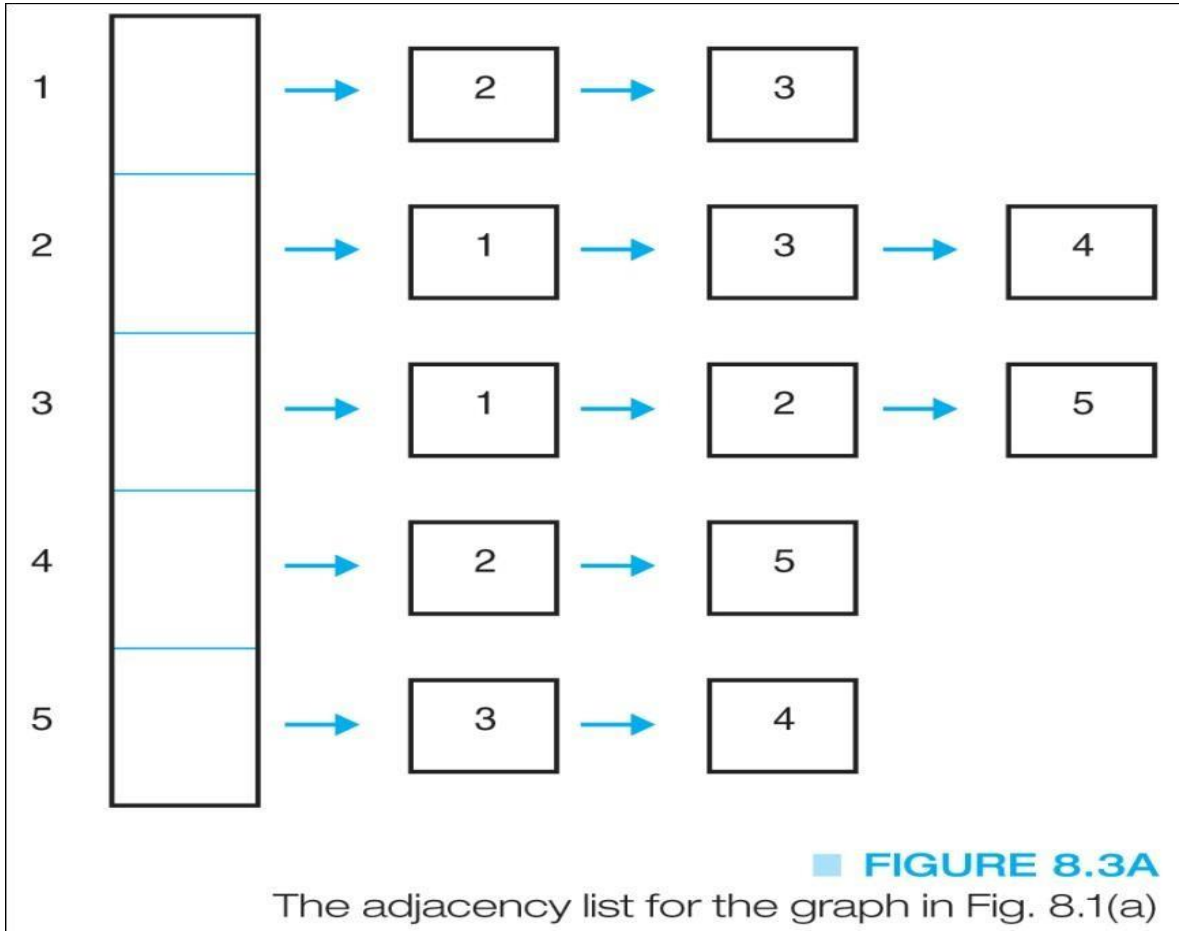


FIGURE 8.3A

The adjacency list for the graph in Fig. 8.1(a)

Adjacency List

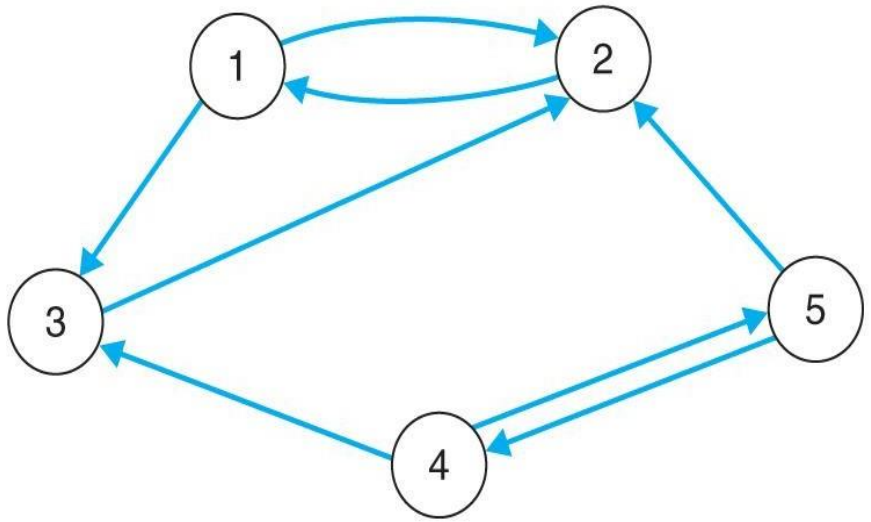


FIGURE 8.1B

The directed graph G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5,4)})

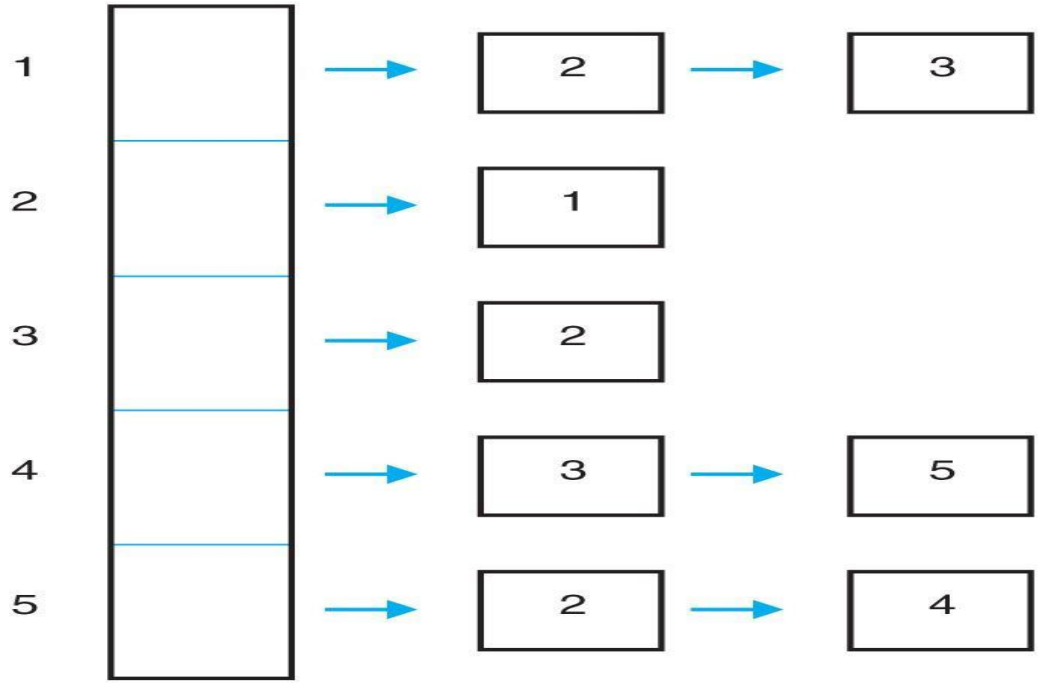


FIGURE 8.3B

The adjacency list for the graph in Fig. 8.1(b)

Graph Traversals

- Some algorithms require that every vertex of a graph be visited exactly once.
- The order in which the vertices are visited may be important, and may depend upon the particular algorithm.
- The two common traversals:
  - depth-first
  - breadth-first

Graph Traversals: Depth First Traversal

- We follow a path through the graph until we reach a dead end.
- We then back up until we reach a node with an edge to an unvisited node.
- We take this edge and again follow it until we reach a dead end.
- This process continues until we back up to the starting node and it has no edges to unvisited nodes.

Depth First Traversal

- Consider the following graph:

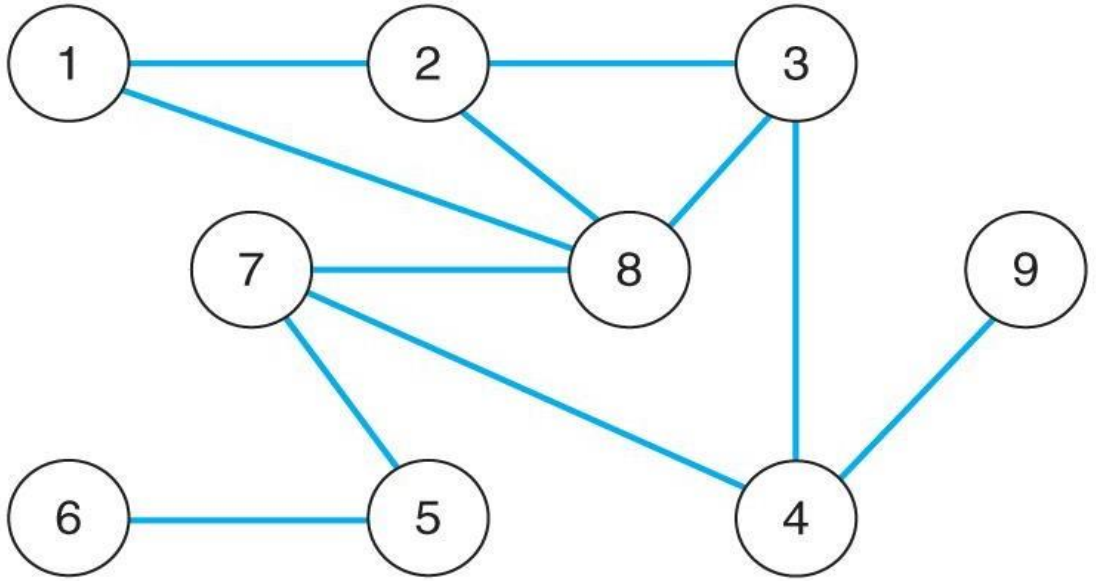


FIGURE 8.4
A graph

- The order of the depth-first traversal of this graph starting at node 1 would be:

1, 2, 3, 4, 7, 5, 6, 8, 9

Graph Traversals: Breadth First Traversal

- From the starting node, we follow all paths of length one.
- Then we follow paths of length two that go to unvisited nodes.
- We continue increasing the length of the paths until there are no unvisited nodes along any of the paths.

Breadth First Traversal Example

- Consider the following graph:

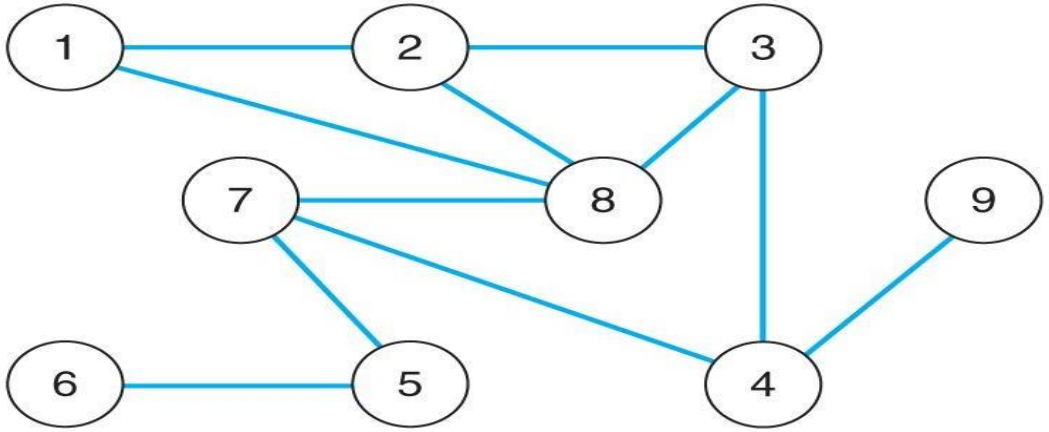


FIGURE 8.4
A graph

- The order of the breadth-first traversal of this graph starting at node 1 would be: 1, 2, 8, 3, 7, 4, 5, 9, 6

Module – 5
Binary Trees and Hashing

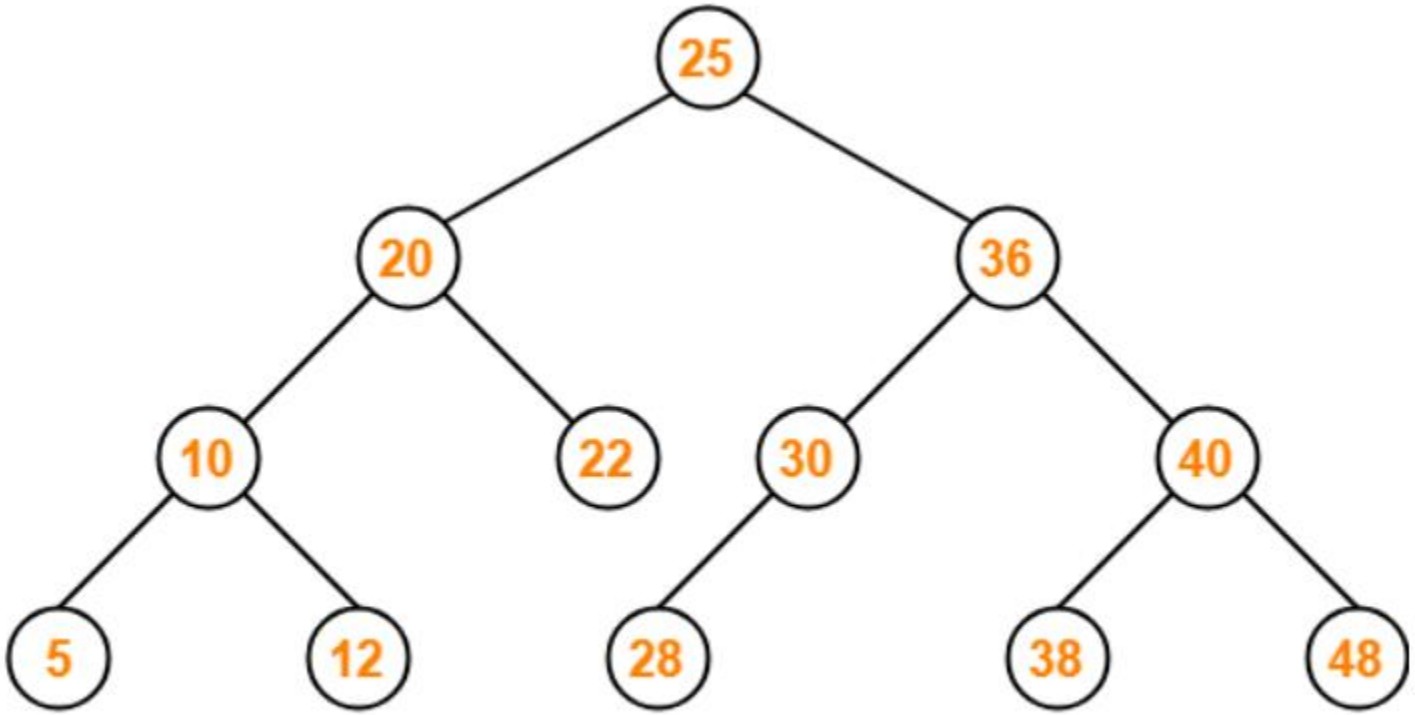
CONTENTS

- Binary Search Trees - Properties and Operations
- Balanced Search Trees – AVL Trees
- M – way Search Trees
- B Trees
- Hashing – Hash Table, Hash Function
- Collisions
- Applications of Hashing

Binary Search Trees (BST)

- In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node n in the tree:
- All keys in n's left sub-tree are less than the key in n, and
- All keys in n's right sub-tree are greater than the key in n.
- In other words, binary search trees are binary trees in which all values in the node’s left sub-tree are less than node value all values in the node’s right sub-tree are greater than node value.

Binary Search Trees (BST)



Binary Search Tree

Binary Search Trees Properties

A BST is a binary tree of nodes ordered in the following way:

i. Each node contains one key (also unique)

ii. The keys in the left sub-tree are < (less) than the key in its parent node

iii. The keys in the right sub-tree > (greater) than the key in its parent node

iv. Duplicate node keys are not allowed.

Inserting a node into BST

- A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root.
- We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node.

Inserting a node into BST

- A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T.
- A definition is as follows.

Insert(N, T) = N if T is empty

= insert(N, T.left) if N < T

= insert(N, T.right) if N > T

Searching a node into BST

- Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

Searching a node into BST

- Search(N, T) = false if T is empty Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).
- A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

Search(N, T) = False if T is empty

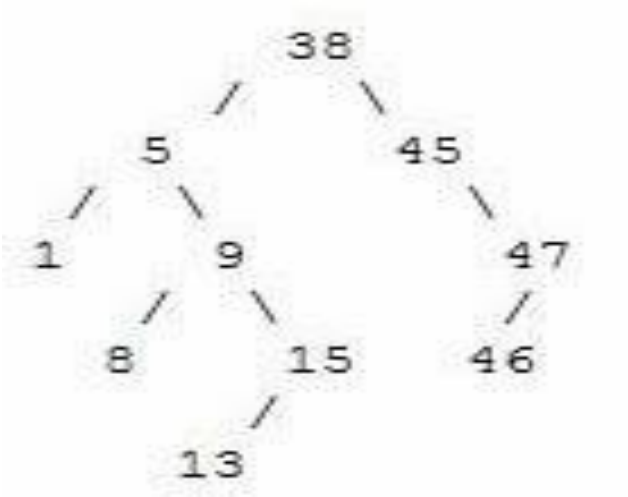
= True if T = N

= search(N, T.left) if N < T

= search(N, T.right) if N > T

Deleting a node into BST

- A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



AVL Trees

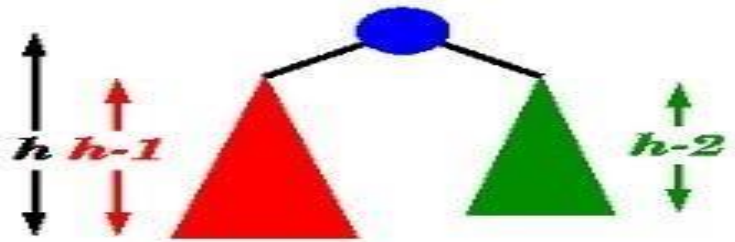
- A **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.
- **AVL Trees:** An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(*log*n)*** search time. Addition and deletion operations also take ***O(*log*n)*** time.

AVL Trees

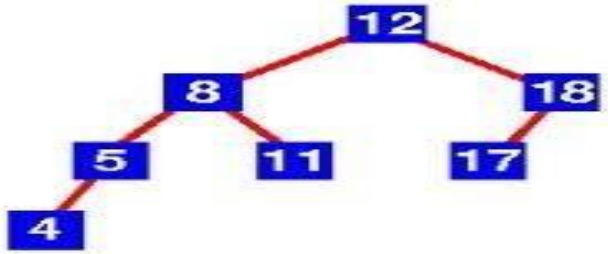
- **Definition of an AVL tree:** An AVL tree is a binary search tree which has the following properties:
  i. The sub-trees of every node differ in height by at most one.
  ii. Every sub-tree is an AVL tree.
- Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

AVL Trees Example

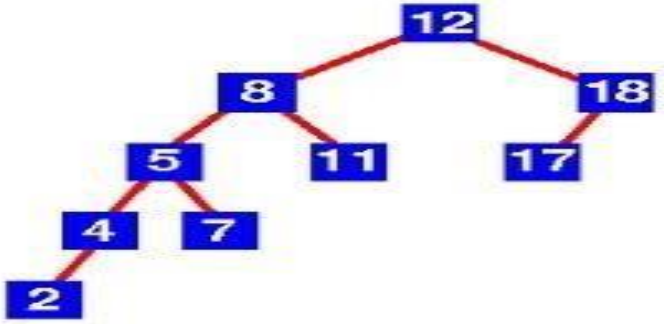
Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.



For example, here are some trees:



Yes this is an AVL tree. Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.



No this is not an AVL tree. Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2.

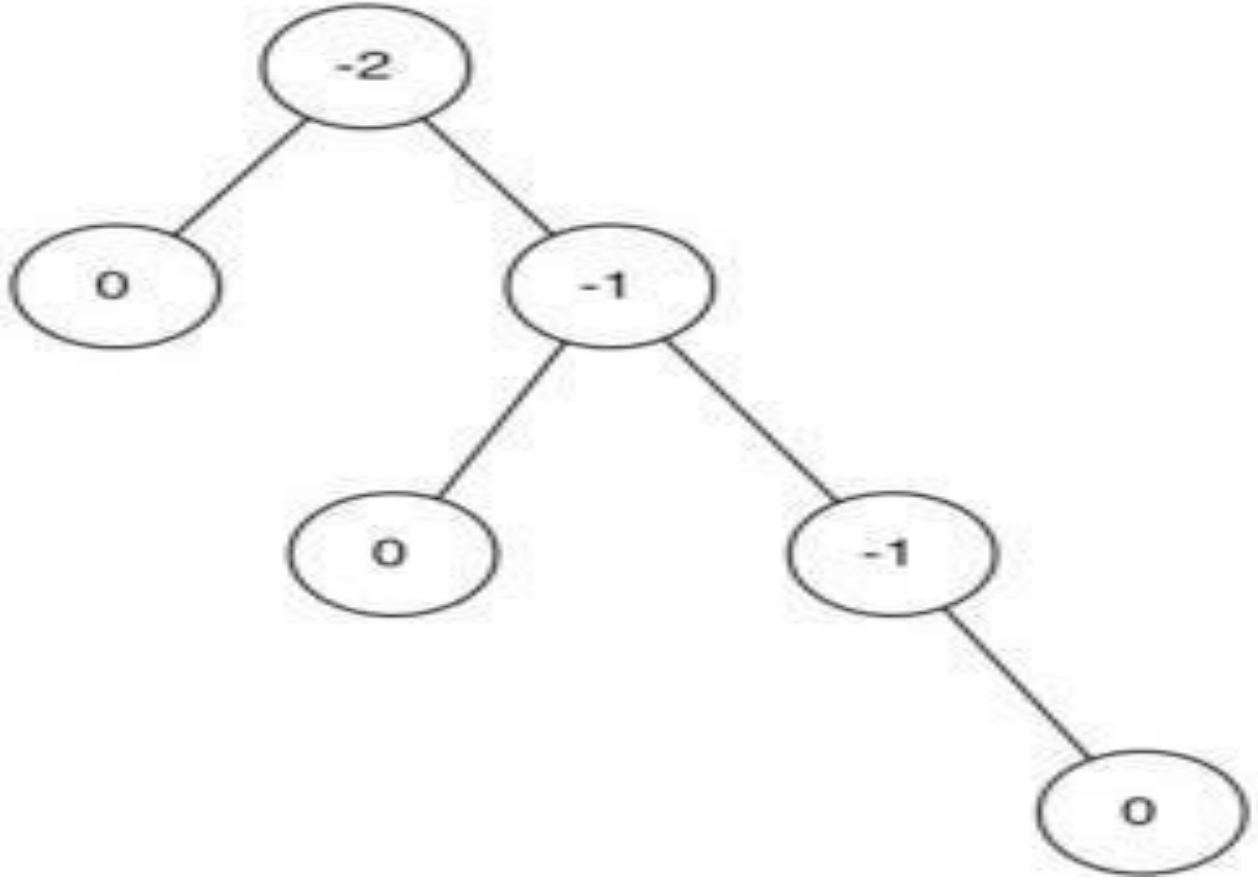
Balance Factor in AVL Trees

- To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

  $balanceFactor = height(leftSubTree) - height(rightSubTree)$

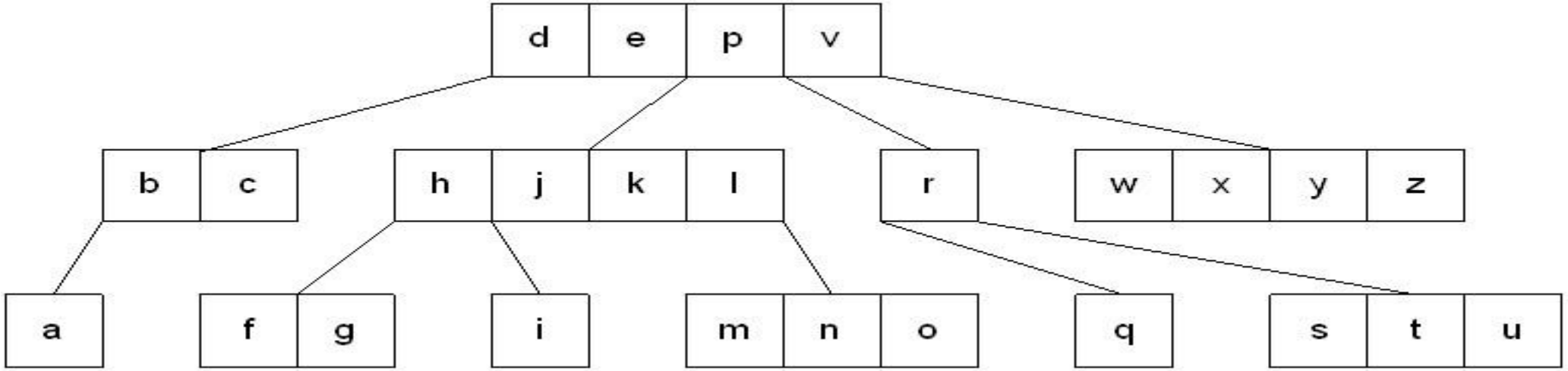
- Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance.

Balance Factor in AVL Trees



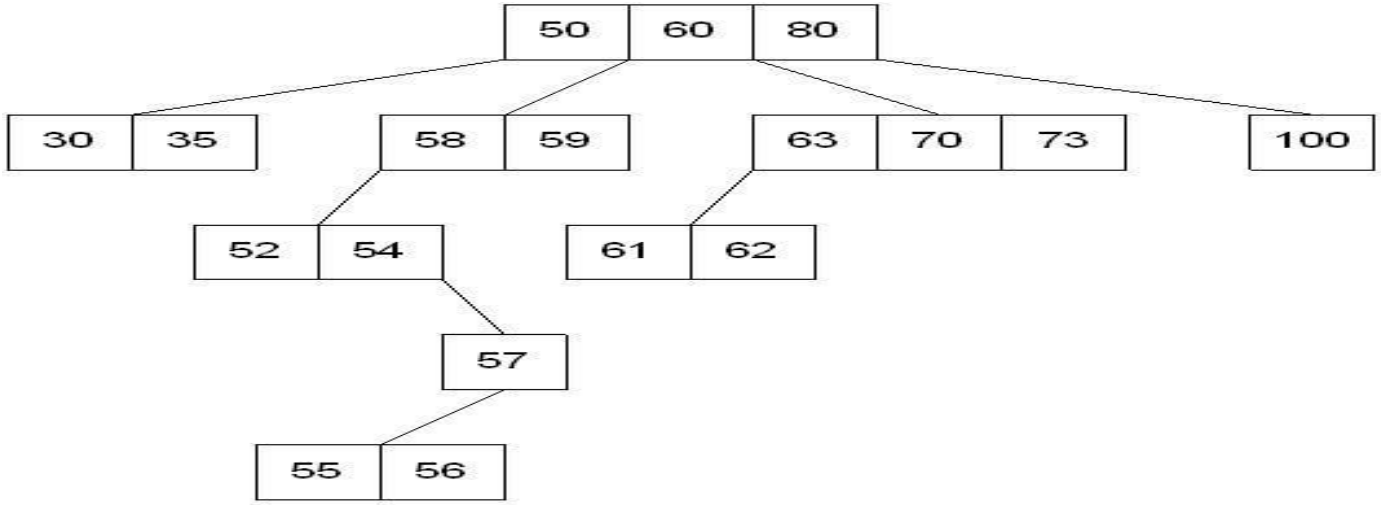
Introduction to M-way Search Trees

- A **multiway tree** is a tree that can have more than two children. A **multiway tree of order m** (or an **m-way tree**) is one in which a tree can have m children.
- As with the other trees that have been studied, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.
- Multiday tree of order 5



Properties of M-way Search Trees

- m-way search tree is a m-way tree in which:
  i. Each node has m children and m-1 key fields
  ii. The keys in each node are in ascending order.
  iii. The keys in the first i children are smaller than the ith key
  iv. The keys in the last m-i children are larger than the ith key
- 4-way search tree



B-Trees

- An extension of a multi-way search tree of order m is a **B- tree of order m**. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.
- A B-tree of order m is a multi-way search tree in which:

i. The root has at least two sub-trees unless it is the only node in the tree.

ii. Each non-root and each non-leaf node have at most m nonempty children and at least m/2 nonempty children.

iii. The number of keys in each non-root and each non-leaf node is One less than the number of its nonempty children.

iv. All leaves are on the same level.

Searching in a B-Trees

- Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node.
- Follow the appropriate pointer to a child node.
- Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

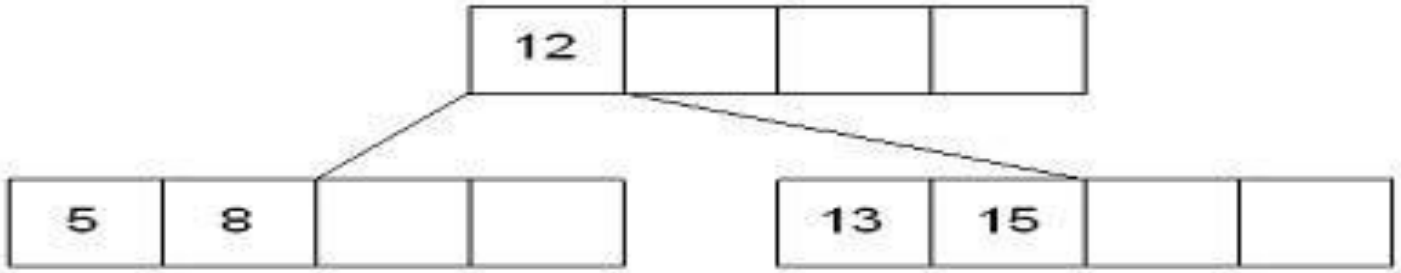
Inserting into a B-Tree

- The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.
- When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:
  i. A key is placed into a leaf that still has room.
  ii. The leaf in which a key is to be placed is full.
  iii. The root of the B-tree is full.

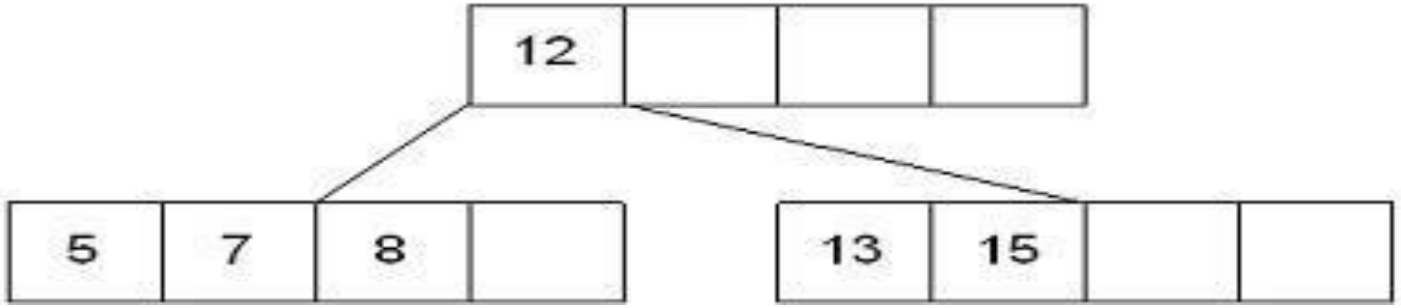
Inserting into a B-Tree

Case 1 : A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:



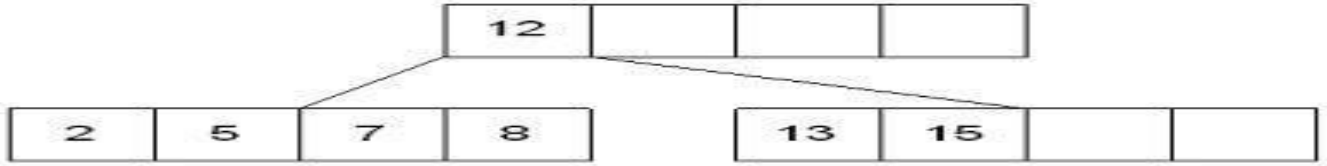
Inserting into a B-Tree

Case 2: The leaf in which a key is to be placed is full

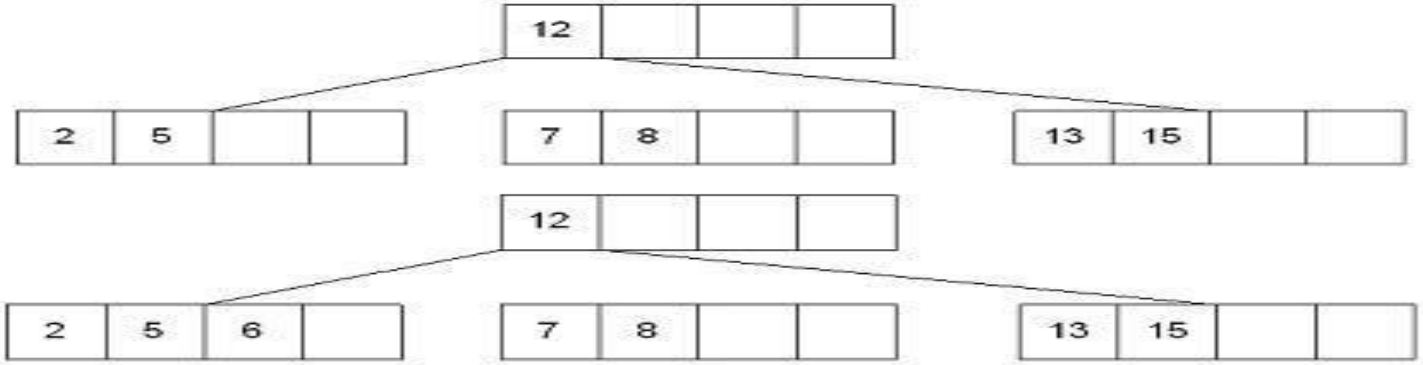
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

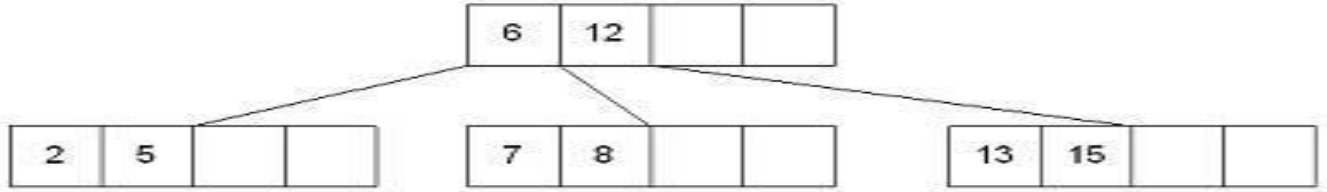
Insert 6 into the following B-tree:



results in a split of the first leaf node:



The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:

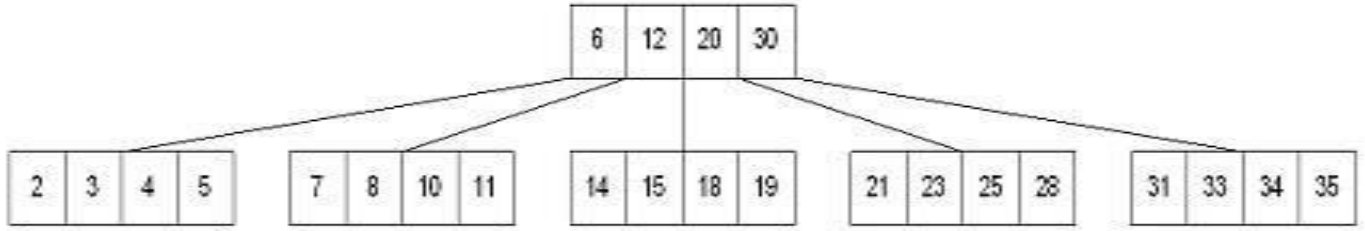


Inserting into a B-Tree

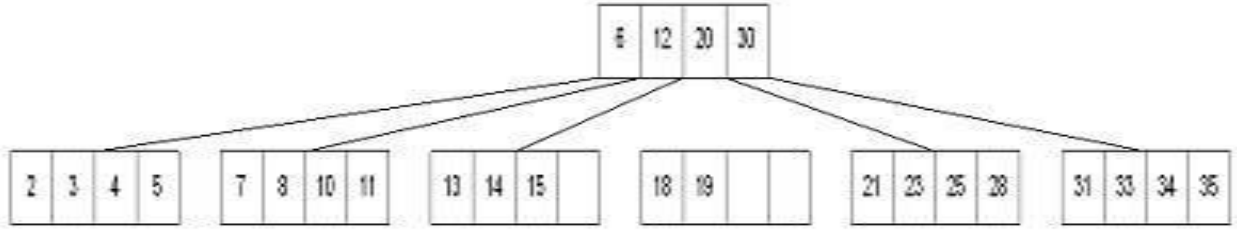
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

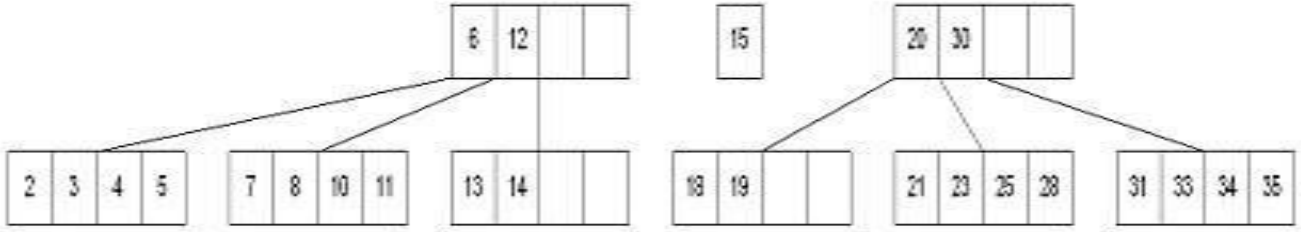
Inserting 13 into the following tree:



Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:

Deletion from a B-Tree

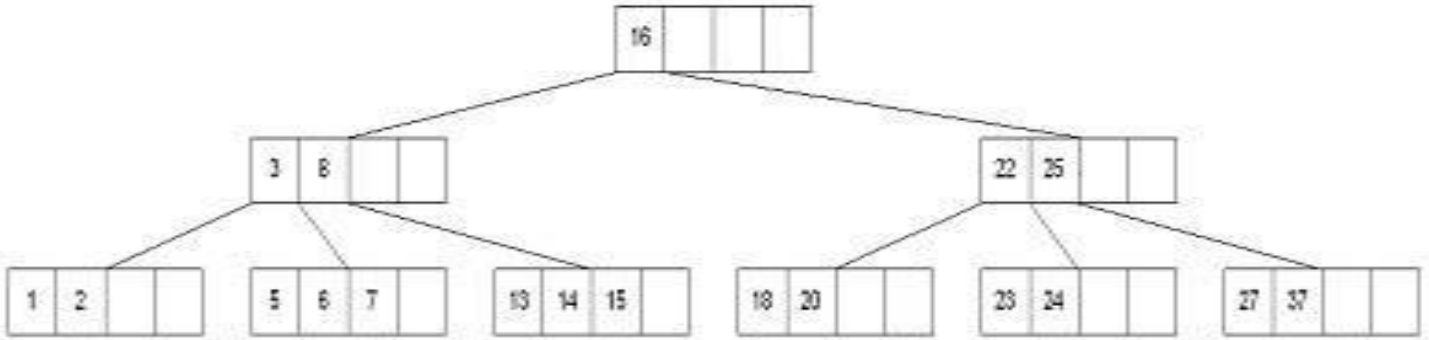
- The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.
- There are two main cases to be considered:
  i. Deletion from a leaf
  ii. Deletion from a non-leaf

Deletion from a B-Tree

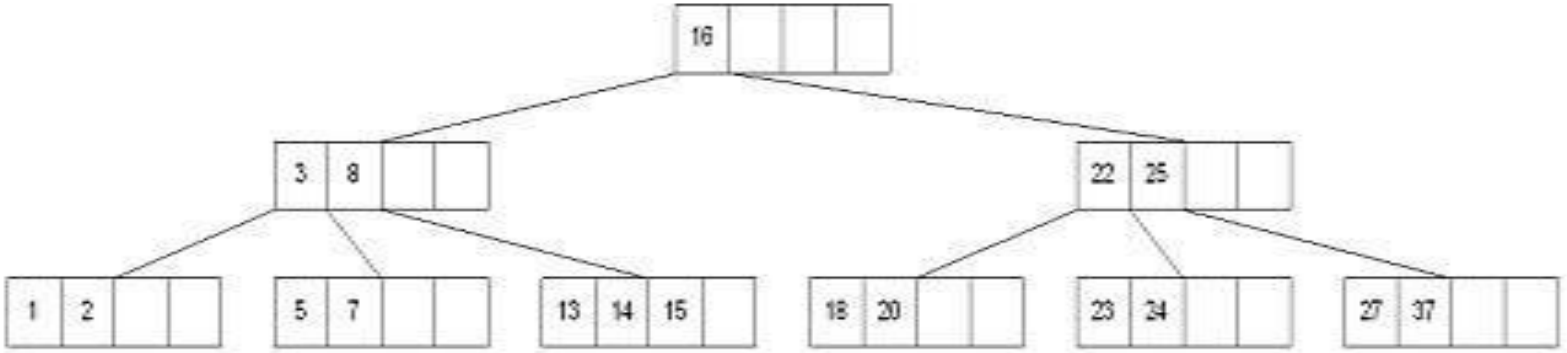
Case 1 : Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

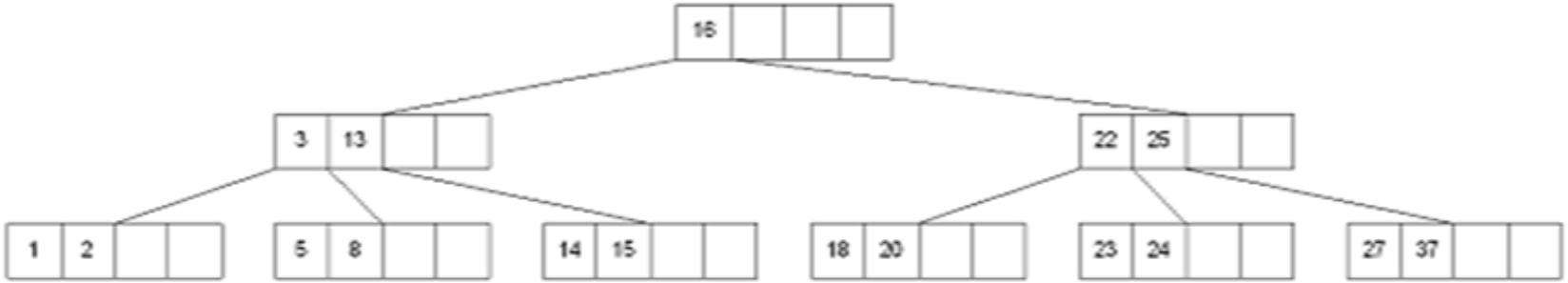
Deleting 6 from the following tree:



results in:



Deletion from a B-Tree



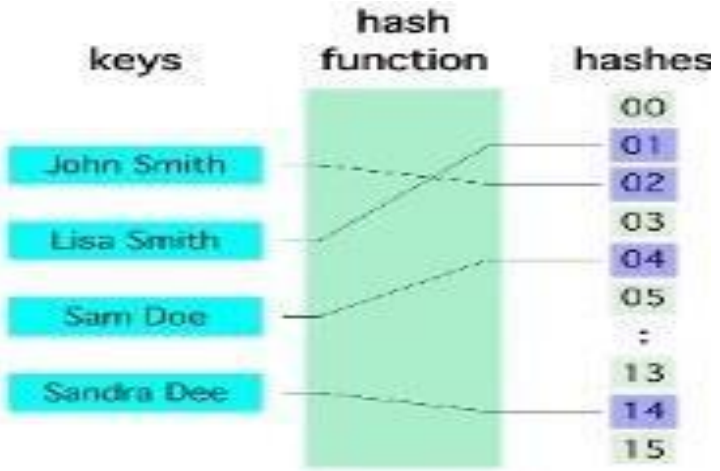
Now delete 8 from the tree:

Hashing

- Hashing is the technique used for performing almost constant time search in case of insertion, deletion and find operation.
- Taking a very simple example of it, an array with its index as key is the example of hash table. So each index (key) can be used for accessing the value in a constant search time. This mapping key must be simple to compute and must helping in identifying the associated value. Function which helps us in generating such kind of key- value mapping is known as Hash Function.
- In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly implemented hash table would always promise an average insert/delete/retrieval time of O(1).

Hashing Function

- A function which employs some algorithm to computes the key *K* for all the data elements in the set *U*, such that the key *K* which is of a fixed size. The same key *K* can be used to map data to a hash table and all the operations like insertion, deletion and searching should be possible. The values returned by a **hash function** are also referred to as **hash** values, **hash** codes, **hash** sums, or **hashes**.



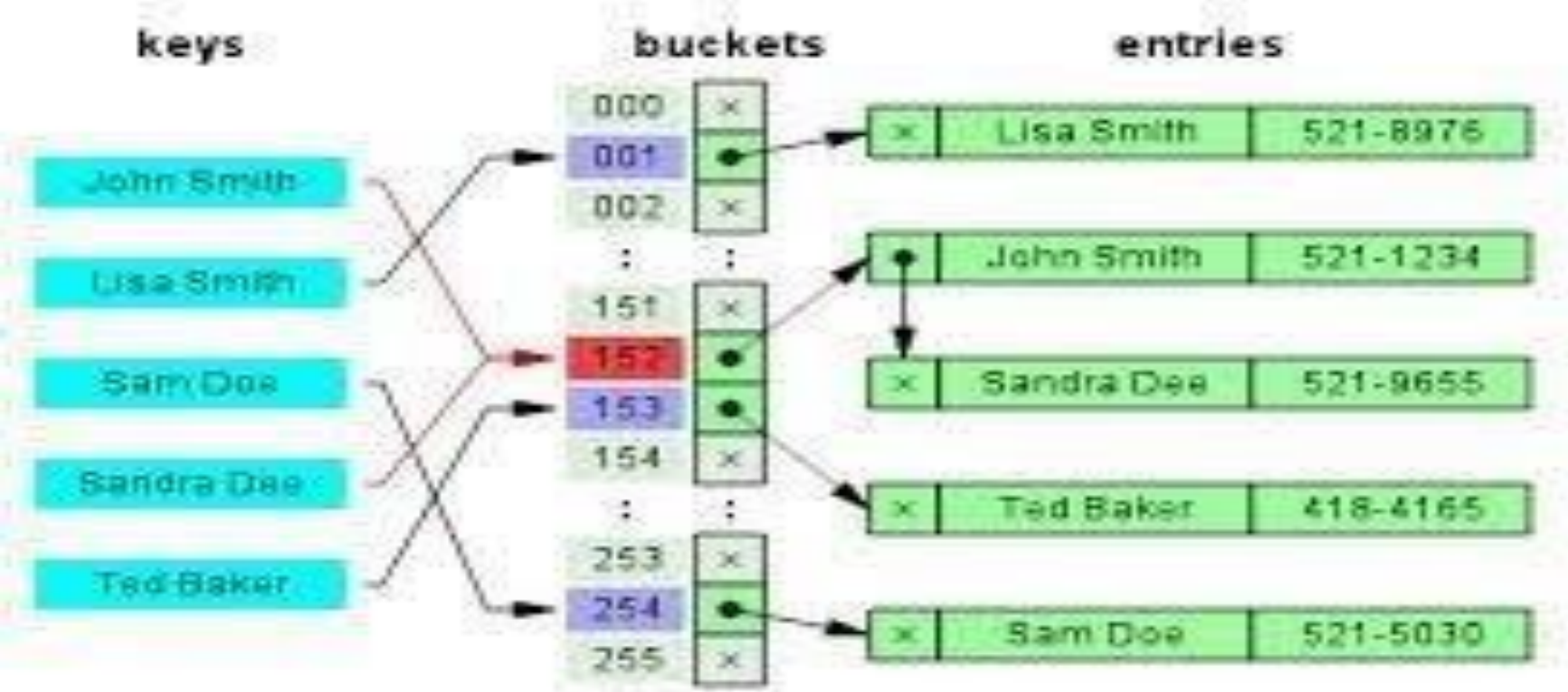
Collision Resolution Techniques

A situation when the resultant hashes for two or more data elements in the data set *U*, maps to the same location in the has table, is called a hash collision. In such a situation two or more data elements would qualify to be stored / mapped to the same location in the hash table.

Hash collision resolution techniques:

• **Open Hashing (Separate chaining):** Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.

Collision Resolution Techniques



Collision Resolution Techniques: Closed Hashing

- In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.
- Liner Probing
- Quadratic probing
- Double hashing

Open Hashing Vs. Closed Hashing

Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space

Applications of Hashing

- A **hash function** maps a variable length input string to fixed length output string **--** its **hash value**, or **hash** for short. If the input is longer than the output, then some inputs must map to the same output **--** a **hash collision**.
- Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, and authentication.
- In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on downloaded files.
- Construct a *message authentication code* (MAC)
- Digital signature
- Make commitments, but reveal message later
- Time-stamping
- Key updating: key is hashed at specific intervals resulting in new key.

THANK YOU